

NetLogo 1.3 User Manual

Table of Contents

<u>What is NetLogo?</u>	1
<u>Features</u>	1
<u>Whats New?</u>	3
<u>Version 1.3 (June 2003)</u>	3
<u>Version 1.2.1 (May 2003)</u>	4
<u>Version 1.2 (March 2003)</u>	4
<u>Version 1.1 (July 2002)</u>	5
<u>Version 1.0 (April 2002)</u>	5
<u>System Requirements</u>	7
<u>System Requirements: Application</u>	7
<u>Windows</u>	7
<u>Macintosh: OS X</u>	7
<u>Macintosh: OS 8.x or 9.x</u>	7
<u>Other platforms</u>	8
<u>System Requirements: Saved Applets</u>	8
<u>Known Issues</u>	9
<u>Known bugs (all systems)</u>	9
<u>Macintosh-only bugs</u>	9
<u>Linux/UNIX-only bugs</u>	10
<u>Known issues with computer HubNet</u>	10
<u>Planned features</u>	10
<u>Unimplemented StarLogoT primitives</u>	10
<u>Contacting Us</u>	13
<u>Website</u>	13
<u>Feedback, Questions, Etc.</u>	13
<u>Reporting Bugs</u>	13
<u>Sample Model: Party</u>	15
<u>At a Party</u>	15
<u>Challenge</u>	17
<u>Thinking With Models</u>	18
<u>Whats Next?</u>	18
<u>Tutorial #1: Models</u>	21
<u>Sample Model: Wolf Sheep Predation</u>	21
<u>Controlling the Model: Buttons</u>	22
<u>Adjusting Settings: Sliders and Switches</u>	23
<u>Gathering Information: Plots and Monitors</u>	25
<u>Plots</u>	25
<u>Monitors</u>	25
<u>Controlling the Graphics Window</u>	25
<u>The Models Library</u>	29
<u>Sample Models</u>	29
<u>Curricular Models</u>	29

Table of Contents

<u>Tutorial #1: Models</u>	
<u>Code Examples</u>	30
<u>HubNet Calculator & Computer Activities</u>	30
<u>Whats Next?</u>	30
<u>Tutorial #2: Commands</u>	31
<u>Sample Model: Traffic Basic</u>	31
<u>The Command Center</u>	31
<u>Working With Colors</u>	34
<u>Agent Monitors and Agent Commanders</u>	36
<u>Whats Next?</u>	39
<u>Tutorial #3: Procedures</u>	41
<u>Setup and Go</u>	41
<u>Patches and Variables</u>	44
<u>An Uphill Algorithm</u>	48
<u>Some More Details</u>	52
<u>Whats Next?</u>	53
<u>Appendix: Complete Code</u>	53
<u>Interface Guide</u>	55
<u>Menus</u>	55
<u>Main Window</u>	57
<u>Interface Tab</u>	57
<u>Procedures Tab</u>	60
<u>Information Tab</u>	61
<u>Errors Tab</u>	62
<u>Programming Guide</u>	63
<u>Agents</u>	63
<u>Procedures</u>	64
<u>Variables</u>	65
<u>Colors</u>	66
<u>Ask</u>	67
<u>Agentsets</u>	68
<u>Breeds</u>	70
<u>Synchronization</u>	71
<u>Procedures (advanced)</u>	71
<u>Lists</u>	72
<u>Strings</u>	75
<u>Turtle shapes</u>	76
<u>HubNet Guide</u>	77
<u>About HubNet</u>	77
<u>HubNet Types</u>	77
<u>What do I need to get started?</u>	78
<u>First-time NetLogo user?</u>	78
<u>Teacher workshops</u>	79

Table of Contents

HubNet Guide

Getting Started With HubNet	79
Using NetLogo	79
HubNet Activities	79
Running an activity	79
HubNet Authoring Guide	81
Computer HubNet Troubleshooting Tips	81
Known Computer HubNet Issues	82
Known bugs (all systems)	83

Shapes Editor Guide.....85

Getting Started	85
Creating and Editing Shapes	85
Using Shapes in a Model	86

BehaviorSpace Guide.....87

What is BehaviorSpace?	87
How It Works	88
Interface Reference	89
Setup Window	89
Progress Dialog	91
Evaluation Window	91

FAQ (Frequently Asked Questions).....95

General	96
Downloading	99
Usage	99
Programming	101

Primitives Dictionary.....107

Categories of Primitives	107
Turtle-related	107
Patch-related primitives	107
Agentset primitives	107
Color primitives	107
Control flow and logic primitives	107
Display primitives	108
HubNet primitives	108
Input/output primitives	108
List primitives	108
String primitives	108
Mathematical primitives	108
Plotting primitives	108
Built-In Variables	108
Turtles	109
Patches	109
Other	109
Keywords	109

Table of Contents

Primitives Dictionary

<u>Constants</u>	109
<u>Mathematical Constants</u>	109
<u>Boolean Constants</u>	109
<u>Color Constants</u>	109
<u>A</u>	110
<u>abs</u>	110
<u>acos</u>	110
<u>and</u>	110
<u>any</u>	110
<u>Arithmetic Operators (+, *, -, /, ^, <, >, =, !=, <=, >=)</u>	111
<u>asin</u>	111
<u>ask</u>	111
<u>at-points</u>	111
<u>atan</u>	112
<u>autoplot?</u>	112
<u>auto-plot-off auto-plot-on</u>	112
<u>B</u>	112
<u>back bk</u>	112
<u>breed</u>	113
<u>breeds</u>	113
<u>but-first bf but-last bl</u>	114
<u>C</u>	114
<u>ceiling</u>	114
<u>clear-all ca</u>	114
<u>clear-all-plots</u>	115
<u>clear-graphics cg</u>	115
<u>clear-output cc</u>	115
<u>clear-patches cp</u>	115
<u>clear-plot</u>	115
<u>clear-turtles ct</u>	116
<u>color</u>	116
<u>cos</u>	116
<u>count</u>	116
<u>create-turtles crt create-<BREED></u>	117
<u>create-custom-turtles cct create-custom-<BREED> cct-<BREED></u>	117
<u>create-temporary-plot-pen</u>	117
<u>D</u>	118
<u>die</u>	118
<u>diffuse</u>	118
<u>diffuse4</u>	119
<u>display</u>	119
<u>distance</u>	119
<u>distance-nowrap</u>	120
<u>distancexy</u>	120
<u>distancexy-nowrap</u>	120
<u>downhill</u>	120
<u>downhill4</u>	121

Table of Contents

Primitives Dictionary

dx dy	121
E	121
empty?	121
end	122
every	122
exp	122
export-output export-plot export-all-plots export-world	122
extract-hsb	123
extract-rgb	124
E	124
filter	124
first	124
floor	124
foreach	125
forward fd	125
fput	125
G	126
get-date-and-time	126
globals	126
H	126
hatch	126
heading	127
hidden?	127
hideturtle ht	127
histogram	127
histogram-list	128
home	128
hsb	128
hubnet-broadcast	129
hubnet-enter-message?	129
hubnet-exit-message?	129
hubnet-fetch-message	129
hubnet-message	129
hubnet-message-source	130
hubnet-message-tag	130
hubnet-message-waiting?	130
hubnet-reset	130
hubnet-send	130
hubnet-set-client-interface	131
I	131
if	131
ifelse	132
import-world	132
in-radius	132
inspect	133
int	133
is-agent? is-agentset? is-boolean? is-list? is-number? is-patch?	

Table of Contents

Primitives Dictionary

	is-patch-agentset? is-string? is-turtle? is-turtle-agentset?	133
	item	134
J		134
	jump	134
L		134
	label	134
	label-color	135
	last	135
	left lt	135
	length	135
	list	136
	ln	136
	locals	136
	log	136
	loop	137
	lput	137
M		137
	map	137
	max	138
	max-one-of	138
	mean	138
	median	138
	member?	138
	min	139
	min-one-of	139
	mod	139
	mouse-down?	139
	mouse-xcor mouse-ycor	140
	myself	140
N		140
	neighbors neighbors4	140
	no-display	141
	no-label	141
	nobody	141
	not	142
	nsum nsum4	142
O		142
	-of	142
	one-of	143
	or	143
	other-turtles-here other-BREED-here	143
P		143
	patch	144
	patch-at	144
	patch-here	144
	patches	144
	patches-own	144

Table of Contents

Primitives Dictionary

pcolor	145
pen-down pd pen-up pu	145
pen-down?	145
plabel	146
plabel-color	146
plot	146
plot-name	146
plot-pen-down ppd plot-pen-up ppu	146
plot-pen-reset	147
plotxy	147
plot-x-min plot-x-max plot-y-min plot-y-max	147
position	147
precision	148
print	148
pxcor	148
pycor	148
R	149
random	149
random-exponential random-normal random-poisson	149
random-n-of	150
random-one-of	150
random-seed	150
read-from-string	151
reduce	151
remainder	152
remove	153
remove-duplicates	153
repeat	153
replace-item	153
report	154
reset-timer	154
reverse	154
rgb	154
right rt	155
round	155
run	155
runresult	155
S	156
scale-color	156
screen-edge-x screen-edge-y	156
screen-size-x screen-size-y	157
self	157
: (semicolon)	157
sentence se	157
set	158
set-current-plot	158
set-current-plot-pen	158

Table of Contents

Primitives Dictionary

set-default-shape	158
set-histogram-num-bars	159
set-plot-pen-color	159
set-plot-pen-interval	159
set-plot-pen-mode	159
set-plot-x-range set-plot-y-range	160
setxy	160
shade-of?	160
shape	160
show	161
showturtle st	161
sin	161
size	161
sort	162
sort-by	162
sprout	162
sqrt	163
stamp	163
standard-deviation	163
startup	163
stop	163
substring	164
sum	164
I	164
tan	164
timer	164
to	164
to-report	165
towards towards-nowrap	165
towardsxy towardsxy-nowrap	165
turtle	166
turtles	166
turtles-at BREED-at	166
turtles-here BREED-here	166
turtles-own BREED-own	167
type	167
U	167
uphill	168
uphill4	168
user-choice	168
user-input	168
user-message	169
user-yes-or-no	169
V	169
value-from	169
values-from	169
variance	170

Table of Contents

Primitives Dictionary

<u>W</u>	170
<u>wait</u>	170
<u>while</u>	170
<u>who</u>	170
<u>with</u>	171
<u>without-interruption</u>	171
<u>word</u>	172
<u>wrap-color</u>	172
<u>X</u>	172
<u>xcor</u>	172
<u>xor</u>	173
<u>Y</u>	173
<u>ycor</u>	173
<u>?</u>	173
<u>?</u>	173

What is NetLogo?

NetLogo is a programmable modeling environment for simulating natural and social phenomena. It is particularly well suited for modeling complex systems developing over time. Modelers can give instructions to hundreds or thousands of independent agents all operating in parallel. This makes it possible to explore the connection between the micro-level behavior of individuals and the macro-level patterns that emerge from the interaction of many individuals.

NetLogo lets students open simulations and play with them, exploring their behavior under various conditions. It is also an authoring tool which enables students, teachers and curriculum developers to create their own models. NetLogo is simple enough that students and teachers can easily run simulations or even build their own. And, it is advanced enough to serve as a powerful tool for researchers in many fields.

NetLogo has extensive documentation and tutorials for all of its features. It also comes with a Models Library, which is a large collection of pre-written simulations that can be used and modified. These simulations address many content areas in the natural and social sciences, including biology and medicine, physics and chemistry, mathematics and computer science, and economics and social psychology. Several model-based inquiry curricula using NetLogo are currently under development.

NetLogo can also power a classroom participatory-simulation tool called HubNet. Through the use of networked computers or handheld devices such as Texas Instruments (TI-83+) calculators, each student can control an agent in a simulation. Follow [this link](#) for more information.

NetLogo is the next generation of the series of multi-agent modeling languages that started with StarLogo. It builds off the functionality of our product [StarLogoT](#) and adds significant new features and a redesigned language and user interface. NetLogo is written in Java so it can run on any operating system (Mac, Windows, Linux, et al). It is run as a standalone application. Individual models can be run as Java applets inside a web browser.

Features

You can use the list below to help familiarize yourself with the features NetLogo has to offer.

- System:
 - ◆ Cross-platform: runs on MacOS, Windows, Linux, et al
 - ◆ Models can be saved as applets to be embedded in web pages
- Language:
 - ◆ Fully programmable
 - ◆ Simple language structure
 - ◆ Language is Logo dialect extended to support agents and parallelism
 - ◆ Unlimited numbers of agents and variables
 - ◆ Double precision platform-independent arithmetic
 - ◆ Many built-in primitives
- Environment:
 - ◆ Interface builder w/ buttons, sliders, switches, choices, monitors, text boxes
 - ◆ "Control strip" including speed slider
 - ◆ Powerful and flexible plotting system

- ◆ Info area for annotating your model
- ◆ HubNet: participatory simulations using networked devices
- ◆ Agent monitors for inspecting and controlling agents
- ◆ BehaviorSpace tool used to collect data from multiple runs of a model
- ◆ Export and import functions (export data, save and restore state of model)

Whats New?

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you. Please send comments, suggestions, and questions to feedback@ccl.northwestern.edu, and bug reports to bugs@ccl.northwestern.edu.

Version 1.3 (June 2003)

- content:
 - ◆ new biology models: Sunflower, Honeycomb
 - ◆ new computer science model: Dining Philosophers
 - ◆ improved models: Stochastic Patchwork (bugfixes, etc.), Equidistant Probability (bugfix, etc.), Wealth Distribution (fixes and improvements), Connected Chemistry models (mostly info tab changes)
- features:
 - ◆ graphics window control strip (including world size adjusters, speed slider, shapes on/off switch, display on/off switch, and edit button)
 - ◆ choice widgets, for presenting the model user with a list of choices
 - ◆ Export World now creates files that follow the CSV standard correctly, so they inter-operate properly with Excel (e.g., you can edit them in Excel and then still re-import them into NetLogo)
 - ◆ the Edit menu now has Comment, Uncomment, Shift Left, and Shift Right items for the Procedures tab: this is handy for commenting out whole blocks of code, and reindenting blocks of code
 - ◆ redesigned Interface toolbar: new icons, removed unnecessary Compile button, removed Halt button (and replaced it with a menu item on the Tools menu) to discourage users from using it to pause models (since halting a model can leave it in an inconsistent state)
 - ◆ on Java 1.3+ VMs only, NetLogo now does all math in Javas strict floating point mode, so you get identical results on all platforms
- language improvements:
 - ◆ new primitives `run` and `runresult` let you run Logo code constructed on the fly
 - ◆ new `map` reporter and `foreach` command let you perform any operation you want on every item in a list
 - ◆ new `filter` primitive lets you remove list items not satisfying a condition
 - ◆ new `reduce` primitive lets you combine the items of a list into a single result
 - ◆ new reporter `sort-by` sorts a list in any order you specify
 - ◆ five primitives (`list`, `word`, `sentence`, `map`, and `foreach`) now accept a variable number of inputs: to use other than the default number, surround the call with parentheses, for example `(list 1)`
 - ◆ added new reporter `self` (so turtles dont have to say `turtle who` anymore; it works for patches too)
 - ◆ added new math reporters `asin` and `acos`
- engine fixes:
 - ◆ fixed bug in `up/downhill(4)` that could cause some winning patches to be ignored if there were multiple winners
 - ◆ sorting lists of strings now works again (had worked in NetLogo 1.1, was broken in 1.2)
 - ◆ fixed bug where under certain circumstances using `ask` inside an explicit or implicit

- without-interruption could cause incorrect behavior when turtles were born, or died, or changed breeds
- ◆ fixed some bugs in Import World involving strings and agentsets
- ◆ `read-from-string` no longer silently ignores invalid input that follows valid input
- ◆ fixed `clear-graphics` so it doesnt clear plots too
- ◆ fixed precedence of the `precision` primitive to be the same as all other prefix reporters
- ◆ changed `scale-color` so that for example `scale-color red 5 0 10` reports 15.0, not 14.99995
- interface fixes:
 - ◆ entering a new id number into a dead turtles turtle monitor now works again
 - ◆ fixed bug where the items in the Procedures menu sometimes werent correctly alphabetized
 - ◆ fixed some cases where you could get a Java exception when trying to edit or import shapes
- HubNet improvements:
 - ◆ many improvements to usability and reliability of computer HubNet
 - ◆ new computer HubNet activity: Sampler
 - ◆ improved activity: Disease (improved function of percentage-infection slider), Tragedy of the Commons (simplified, improved usability)
 - ◆ most HubNet models can now run even when HubNet itself isnt running
 - ◆ `hubnet-set-tags` removed from the language; its functionality is now merged into `hubnet-set-client-interface`; `hubnet-set-client-interface` has changed to take a string and a list, and must be called before `hubnet-reset`; see the Primitives Dictionary and HubNet Guide for more information

Version 1.2.1 (May 2003)

- documentation:
 - ◆ expanded FAQ
- engine fixes:
 - ◆ fixed 1.2-only bug where sometimes turtles would fail to draw after you changed the world size
 - ◆ fixed compiler bug where using `downhill` and other primitives with variable other than a patch variable would cause a Java exception
 - ◆ saved applets can now use `import-world`
- interface fixes:
 - ◆ fixed 1.2-only bug where running some models in BehaviorSpace would slow them down (this had been fixed before, but only partially)
 - ◆ fixed Linux/Unix-only, 1.2-only bug where the agent type selector in the command center didnt work

Version 1.2 (March 2003)

- models run faster now
- beta-level support for running as a native application on Mac OS X
- running the full authoring environment as an applet in a web browser is no longer supported (saving individual models as applets is still supported, however)
- alpha release of computer HubNet: formerly HubNet required the TI Navigator calculator network to operate; now you can use it over TCP/IP with networks of laptop or desktop

computers

- many new primitives and other language improvements
- display of coordinates when mousing over plots

Version 1.1 (July 2002)

- models run faster now
- Save as Applet lets you embed your model in any web page
- printer support
- Procedures menu
- scrollable Interface tab
- contextual menus in Interface tab
- improved agent monitors
- experimental Turtle Sizes and Exact Turtle Positions options
- many new primitives
- improved HubNet support, improved activities

Version 1.0 (April 2002)

- initial release (after a series of betas)

System Requirements

NetLogo is designed to run almost any type of computer, but you may experience problems with older, less powerful systems or older versions of system software. The exact requirements are summarized below. If you have any trouble with NetLogo not working on your system, we would like to offer assistance. Please write bugs@ccl.northwestern.edu.

System Requirements: Application

On any type of computer, a Java Virtual Machine that supports Java version 1.1 or later is required. (Note that on the Windows platform, Microsofts Java Virtual Machine is not supported.) On the Windows, Macintosh, and Linux platforms, you can choose to include the Java Virtual Machine with the download.

On all systems, approximately 15MB of free hard drive space is required.

Windows

- Windows 95, NT, 98, ME, 2000, or XP (note: we dont actively test 95 or NT ourselves, but we havent received problem reports from users on those systems)
- 64 MB RAM (or probably more for NT/2000/XP)

Note: Microsofts Java Virtual Machine (jview.exe) is not supported.

Macintosh: OS X

- OS X version 10.1.3 or later
- 128 MB RAM (256 MB RAM recommended)

On OS X, a Java Virtual Machine is included as part of the operating system. You do not need to obtain one separately. If you are running OS X 10.1, make sure you have installed Java 1.3.1 Update 1, available through Software Update.

If you install NetLogo on OS X, both a native and a Classic application will be installed. The native application is still in beta. It is beta because running natively is mostly reliable, but there are some bugs. (See [Known Issues](#) for a list of problems you may encounter with the native application.) So for the time being, if you need maximum reliability you may wish to run under Classic instead.

Macintosh: OS 8.x or 9.x

- PowerPC processor
- OS version 8.6 (other 8.x versions may also work; 8.6 is the version we actually test)
- 64 MB RAM (96 MB RAM recommended)

Youll need Apples Macintosh Runtime for Java: [MRJ 2.2.5](#) or higher (2.2.3 and 2.2.4 will probably also work). You can request that this VM be included at the time you download NetLogo.

Other platforms

NetLogo should work on any platform on which a Java Virtual Machine, version 1.1 or later, is available. If you have trouble, please contact us (see above).

System Requirements: Saved Applets

Saved applets should work on any web browser and platform on which a Java Virtual Machine, version 1.1 or later, is available. If you have trouble, please contact us (see above).

Exception: Apples new Java 1.4.1 virtual machine on Mac OS X does not run saved applets well. Some OS X web browsers, for example Internet Explorer 5.2.2, still use the old Java 1.3.1 virtual machine, which works better. Other browsers, for example Safari, are already using the new VM. We do plan to make saved applets function better in the new VM; in the meantime, you may wish to view saved applets in a browser which still uses the old VM. (To see which VM a saved applet is running under, hold down the Control key and click the mouse in the white background of the applet.)

Known Issues

If NetLogo malfunctions, please send us a bug report. See the [Contact Us](#) section for instructions.

Known bugs (all systems)

- The Turtle Sizes and Exact Turtle Positions features can cause display anomalies under some circumstances
- Integers in NetLogo must lie in the range -2147483648 to 2147483647 ; if you exceed this range, instead of a runtime error occurring, you get incorrect results
- Printing the Interface tab doesn't work perfectly on every Java VM
- Out-of-memory conditions are not handled gracefully

Macintosh-only bugs

On all versions of MacOS:

- Double clicking a selected interface item to edit it doesn't work; use the Edit button instead, or control-click and choose Edit from the contextual menu
- The User Manual item on the Help menu will sometimes launch a web browser other than your default browser

When running OS 8, OS 9, or in Classic on OS X:

- The Help menu has an extra blank item in it

When running in Classic on OS X:

- When running in Classic under Mac OS X, your model can't use more than about 100MB of RAM (or less if other Classic applications are running). This is due to a limitation of the Classic environment itself. If you need to run very large models on Mac OS X, then you should use the native application, not the Classic application.

When running natively on OS X: (because of the following issues, our OS X native support is considered beta; for maximum reliability, you may wish to run NetLogo under Classic instead)

- NetLogo may intermittently and unpredictably freeze or crash; this happens more frequently on machines with dual processors
- The output portion of the command center doesn't always scroll all the way to the bottom when new output is added, but stops one line short
- The `mouse-down?`, `mouse-xcor`, and `mouse-ycor` primitives sometimes become confused; moving the mouse into the graphics window and then out again should clear it up
- When editing the contents of the Interface tab, NetLogo can sometimes become confused about what widgets are selected; clicking in the background of the tab usually clears this up
- Button presses may occasionally cause Java exceptions; this seems to happen most often on dual processor machines, and in models that run code in multiple buttons simultaneously (for example Tetris)
- The command center intermittently is the wrong size

- Some models that use the experimental turtle sizes and exact turtle positions features may have excessive flicker in the graphics window
- Turtle and patch labels may intermittently not be drawn correctly in some models, particularly when using the Zoom in and Zoom out menu items
- Sometimes the line spacing in the info tab is wildly too large; quitting and restarting NetLogo usually fixes the problem
- In some modal dialog boxes, the OK button fails to operate; you may close the dialog using the red close button at the left of the dialogs title bar
- An agent monitor cant be dragged to a new location without first clicking on it to make it the front window

Linux/UNIX-only bugs

- Resizing and moving of items in the Interface tab functions only erratically

Known issues with computer HubNet

See the [HubNet Guide](#) for a list of known issues with computer HubNet.

Planned features

This is only a partial list of features we plan to add. Periodically, new versions of NetLogo will be available from our web site.

- Optional randomized agent scheduler
- Ability to have multiple models open
- Making a QuickTime movie of your model
- Java API for user-defined extensions to NetLogo
- Improved support for HubNet
- Parenthesis and bracket matching in the editor

Unimplemented StarLogoT primitives

The following StarLogoT primitives are not available in NetLogo. (Note that many StarLogoT primitives, such as `count-turtles-with`, are intentionally not included in this list because NetLogo allows for the same functionality with the new `agentset` syntax.)

- `ifelse-report`
- `maxint`, `minint`, `maxnum`, `minnum`
- `beep`, `readlist`
- `import-turtles`, `import-patches`, `import-turtles-and-patches` (note that NetLogo adds `import-world`, though)
- miscellaneous seldom-used plotting reporters such as `plot-pencolor`, `pp-plotlist`, `pp-plotpointlist`, `ppinterval`, `ppxcor`, `ppycor`, etc.
- `bit`, `bitand`, `bitneg`, `bitor`, `bitset`, `bitstring`, `bitxor`, `make-bitarray`, `rotate-left`, `rotate-right`, `shift-left`, `shift-right`
- `close-movie`, `open-movie/setup-movie`, `movie-snapshot`, `snapshot`
- `load-pict`, `save-pict`
- `camera-brightness`, `camera-click`, `camera-init`, `camera-set-brightness`

- `choose-directory-dialog`, `choose-file-dialog`, `choose-new-file-dialog`,
`netlogo-directory`, `project-directory`, `project-name`, `project-pathname`,
`save-project`

Contacting Us

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you.

Website

Our website at ccl.northwestern.edu includes our mailing address and phone number. It also has information about our staff and our various research activities.

Feedback, Questions, Etc.

If you have general feedback, suggestions, or questions, write to feedback@ccl.northwestern.edu.

Reporting Bugs

If you would like to report a bug that you find in NetLogo, write to bugs@ccl.northwestern.edu. When submitting a bug report, please try to include as much of the following information as possible:

- A complete description of the problem and how it occurred.
- The NetLogo model or code you are having trouble with. If possible, attach a complete model.
- Your system information: NetLogo version, OS version, and so on. This information is available from NetLogo's About NetLogo menu item.
- Any error messages that were displayed.

Sample Model: Party

This activity is designed to get you thinking about computer modeling and how you can use it. It also gives you some insight into the NetLogo software. We encourage beginning users to start with this activity.

At a Party

Have you ever been at a party and noticed how people cluster in groups? You may have also noticed that people do not stay within one group, but move throughout the party. As individuals move around the party, the groups change. If you watched these changes over time, you would notice patterns forming.

For example, in social settings, people tend to exhibit different behavior than when they are at work or home. Individuals who are confident within their work environment may become shy and timid at a social gathering. And others who are quiet and reserved at work may be the party starter with friends.

The patterns may also depend on what kind of gathering it is. In some settings, people are trained to organize themselves into mixed groups; for example, party games or school-like activities. But in a non-structured atmosphere, people tend to group in a more random manner.

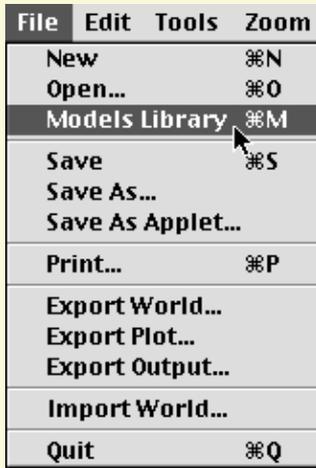
Is there any type of pattern to this kind of grouping?

Lets take a closer look at this question by using the computer to model human behavior at a party. NetLogos Party model looks specifically at the question of grouping by gender at parties: why do groups tend to form that are mostly men, or mostly women?

Lets use NetLogo to explore this question.

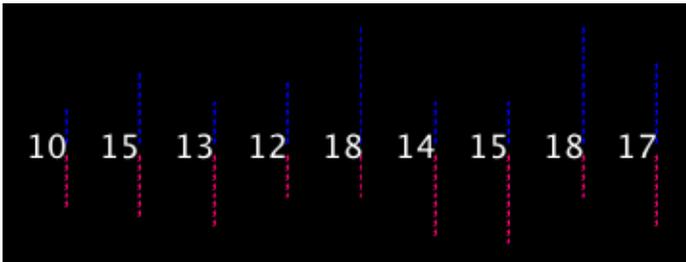
What to do:

1. Start NetLogo.
2. Choose Models Library from the File menu.



3. Open the Social Science folder.
4. Click on the model called Party.
5. Press the open button.
6. Wait for the model to finish loading
7. (optional) Make the NetLogo window bigger so you can see everything.
8. Press the setup button.

In the Graphics Window, you will see pink and blue lines with numbers:



These lines represent mingling groups at a party. Men are represented in blue, women in pink. The numbers are the total number of people in each group.

Do all the groups have about the same number of people?

Do all the groups have about the same number of each sex?

Lets say you are having a party and invited 150 people. You are wondering how people will gather together. Suppose 10 groups form at the party.

How do you think they will group?

Instead of asking 150 of your closest friends to gather and randomly group, lets have the computer simulate this situation for us.

What to do:

1. Press the go button. (Pressing go again will stop the model manually.)
2. Observe the movement of people until the model stops.
3. Watch the plots to see whats happening in another way.

Now how many people are in each group?

Originally, you may have thought 150 people splitting into 10 groups, would result in about 15 people in each group. From the model, we see that people did not divide up evenly into the 10 groups instead, some groups became very small, whereas other groups became very large. Also, the party changed over time from all mixed groups of men and women to all single-sex groups.

What could explain this?

There are lots of possible answers to this question about what happens at real parties. The designer of this simulation thought that groups at parties dont just form randomly. The groups are determined by how the individuals at the party behave. The designer chose to focus on a particular variable, called tolerance:



Tolerance is defined here as the percentage of people of the opposite sex an individual is comfortable with. If the individual is in a group that has a higher percentage of people of the opposite sex than their tolerance allows, then they become uncomfortable and leave the group to find another group.

For example, if the tolerance level is set at 25%, then males are only comfortable in groups that are less than 25% female, and females are only comfortable in groups that are less than 25% male.

As individuals become uncomfortable and leave groups, they move into new groups, which may cause some people in that group to become uncomfortable in turn. This chain reaction continues until everyone at the party is comfortable in their group.

Note that in the model, tolerance is not fixed. You, the user, can use the tolerance slider to try different tolerance percentages and see what the outcome is when you start the model over again.

How to start over:

1. If the go button is pressed (black), then the model is still running. Press the button again to stop it.
2. Adjust the tolerance slider to a new value by dragging its red handle.
3. Press the setup button to reset the model.
4. Press the go button to start the model running again.

Challenge

As the host of the party, you would like to see both men and women mingling within the groups. Adjust the tolerance slider on the side of the Graphics Window to get all groups to be mixed as an end result.

To make sure all groups of 10 have both sexes, at what level should we set the tolerance?

Test your predictions on the model.

Can you see any other factors or variables that might affect the male to female ratio within each group?

Make predictions and test your ideas within this model. Feel free to manipulate more than one variable at a time.

As you are testing your hypotheses, you will notice that patterns are emerging from the data. For example, if you keep the number of people at the party constant but gradually increase the tolerance level, more mixed groups appear.

How high does the tolerance value have to be before you get mixed groups?

What percent tolerance tends to produce what percentage of mixing?

Thinking With Models

Using NetLogo to model situations like this party scenario allows you to experiment with a system in a rapid and flexible way that would be difficult to do in a real world situation. Modeling also gives you the opportunity to observe a situation or circumstance with less prejudice as you can examine the underlying dynamics of a situation. You may find that as you model more and more, many of your preconceived ideas about various phenomena will be challenged. For example, a surprising result of the Party model is that even if tolerance is relatively high, a great deal of separation between the sexes occurs.

This is a classic example of an emergent phenomenon, where a group pattern results from the interaction of many individuals. This idea of emergent phenomena can be applied to almost any subject.

What other emergent phenomena can you think of?

To see more examples and gain a deeper understanding of this concept and how NetLogo helps learners explore it, you may wish to explore NetLogos Models Library. It contains models that demonstrate these ideas in systems of all kinds.

For a longer discussion of emergence and how NetLogo helps learners explore it, see [Modeling Natures Emergent Patterns with Multi-agent Languages](#) (Wilensky, 2001).

Whats Next?

The section of the User Manual called [Tutorial #1: Running Models](#) goes into more detail about how to use the other models in the Models Library.

If you want to learn how to explore the models at a deeper level, [Tutorial #2: Commands](#) will introduce you to the NetLogo modeling language.

Eventually, you'll be ready for [Tutorial #3: Procedures](#), where you can learn how to alter and extend existing models to give them new behaviors, and build your own models.

Tutorial #1: Models

If you read the [Sample Model: Party](#) section, you got a brief introduction to what its like to interact with a NetLogo model. This section will go into more depth about the features that are available while youre exploring the models in the Models Library.

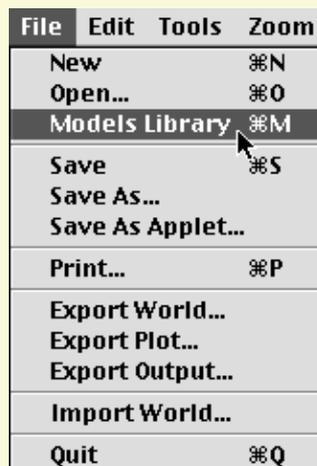
Throughout all of the tutorials, well be asking you to make predictions about what the effects of making changes to the models will be. Keep in mind that the effects are often surprising. We think these surprises are exciting and provide excellent opportunities for learning.

Some people have found it helpful to print out the tutorials in order to work through them. When the tutorials are printed out, theres more room on your computer screen for the NetLogo model youre looking at.

Sample Model: Wolf Sheep Predation

Well open one of the Sample Models and explore it in detail. Lets try a biology model: Wolf Sheep Predation, a predator–prey population model.

- Open the Models Library from the File menu.



- Choose Wolf Sheep Predation from the Biology section and press Open.

The Interface tab will fill up with lots of buttons, switches, sliders and monitors. These interface elements allow you to interact with the model. Buttons are blue; they set up, start, and stop the model. Sliders and switches are green; they alter model settings. Monitors and plots are beige; they display data.

If youd like to make the window larger so that everything is easier to see, you can use the zoom menu at the top of the window.

When you first open the model, you will notice that the Graphics Window is empty (all black). To begin the model, you will first need to set it up.

- Press the setup button.

What do you see appear in the Graphics Window?

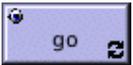
- Wait until the setup button pops back up (in other words, turns blue again).
- Press the go button to start the simulation.

As the model is running, what is happening to the wolf and sheep populations?

- Press the go button to stop the model.

Controlling the Model: Buttons

When a button is pressed, the model responds with an action. A button can be a once button, or a forever button. You can tell the difference between these two types of buttons by a symbol on the face of the button. Forever buttons have two arrows in the bottom right corners, like this:



Once buttons don't have the arrows, like this:



Once buttons do one action and then stop. When the action is finished, the button pops back up. (Note: If you do not wait for the button to pop back up before pressing any other buttons, the model might get confused, and you may get an error message.)

Forever buttons do an action over and over again. When you want the action to stop, press the button again. It will finish the current action, then pop back up.

Most models, including Wolf Sheep Predation, have a once button called setup and a forever button called go. Many models also have a once button called go once or step once which is like go except that it advances the model by one time step instead of over and over. Using a once button like this lets you watch the progress of the model more closely.

Stopping a forever button is the normal way to stop a model. It's safe to pause a model by stopping a forever button, then make it go on by pressing the button again. You can also stop a model with the Halt item on the Tools menu, but you should only do this if the model is stuck for some reason. Using Halt may interrupt the model in the middle of an action, and as the result the model could get confused.

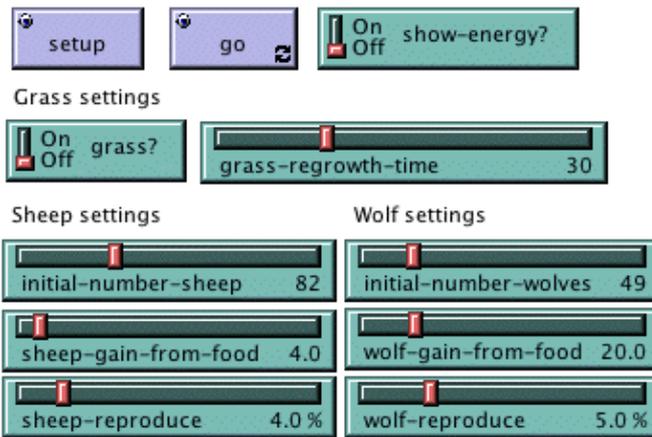
- If you like, experiment with the setup and go buttons in the Wolf Sheep Predation model.

Do you ever get different results if you run the model several times with the same settings?

Adjusting Settings: Sliders and Switches

The settings within a model give you an opportunity to work out different scenarios or hypotheses. Altering the settings and then running the model to see how it reacts to those changes can give you a deeper understanding of the phenomena being modeled. Switches and sliders give you access to a models settings.

Here are the switches and sliders in Wolf Sheep Predation:



Lets experiment with their effect on the behavior of the model.

- Open Wolf Sheep Predation if its not open already.
- Press setup and go and let the model run for about a 100 time-ticks. (Note: there is a readout of the number of ticks right above the plot.)
- Stop the model by pressing the go button.

What happened to the sheep over time?

Lets take a look and see what would happen to the sheep if we change one of the settings.

- Turn the grass? switch on.
- Press setup and go and let the model run for a similar amount of time as before.

What did this switch do to the model? Was the outcome the same as your previous run?

Just like buttons, switches have information attached to them. Their information is set up in an on/off format. Switches turn on/off a separate set of directions. These directions are usually not necessary for the model to run, but might add another dimension to the model. Turning the grass? switch on

affected the outcome of the model. Prior to this run, the growth of the grass stayed constant. This is not a realistic look at the predator–prey relationship; so by setting and turning on a grass growth rate, we were able to model all three factors: sheep, wolf and grass populations.

Another type of setting is called a slider.

Sliders are a different type of setting than a switch. A switch has two values: on or off. A slider has a range of numeric values that can be adjusted. For example, the initial–number–sheep slider has a minimum value of 0 and a maximum value of 250. The model could run with 0 sheep or it could run with 250 sheep, or anywhere in between. Try this out and see what happens. As you move the marker from the minimum to the maximum value, the number on the right side of the slider changes; this is the number the slider is currently set to.

Lets investigate Wolf Sheep Predations sliders.

- Read the contents of the Information tab, located above the toolbar, to learn what each of this models sliders represents.

The Information tab is extremely helpful for gaining insight into the model. Within this tab you will find an explanation of the model, suggestions on things to try, and other information. You may want to read the Information tab before running a model, or you might want to just start experimenting, then look at the Information tab later.

What would happen to the sheep population if there was more initial sheep and less initial wolves at the beginning of the simulation?

- Turn the grass? switch off.
- Set the initial–number–sheep slider to 100.
- Set the initial–number–wolves slider to 20.
- Press setup and then go.
- Let the model run for about 100 time–ticks.

Try running the model several times with these settings.

What happened to the sheep population?

Did this outcome surprise you? What other sliders or switches can be adjusted to help out the sheep population?

- Set initial–number–sheep to 80 and initial–number–wolves to 50. (This is close to how they were when you first opened the model.)
- Set sheep–reproduce to 10.0%.
- Press setup and then go.
- Let the model run for about 100 time ticks.

What happened to the wolves in this run?

When you open a model, all the sliders and switches are on a default setting. If you open a new model or exit the program, your changed settings will not be saved, unless you choose to save

them.

(Note: in addition to sliders and switches, some models have a third kind of setting, called a choice. The Wolf Sheep Predation doesnt have any of these, though.)

Gathering Information: Plots and Monitors

A purpose to modeling is to gather data on a subject or topic that would be very difficult to do in a laboratory situation. NetLogo has two main ways of displaying data to the user: plots and monitors.

Plots

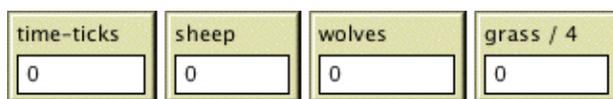
The plot in Wolf Sheep Predation contains three lines: sheep, wolves, and grass / 4. (The grass count is divided by four so it doesnt make the graph too tall.) The lines show whats happening in the model over time. To see which line is which, click on Pens in the upper right corner of the plot window to open the plot pens legend. A key appears that indicates what each line is plotting. In this case, its the population counts.

When a plot gets close to becoming filled up, the horizontal axis increases in size and all of the data from before gets squeezed into a smaller space. In this way, more room is made for the plot to grow.

If you want to save the data from a plot to view or analyze it in another program, you can use the Export Plot item on the File menu. It saves this information to your computer in a format that can be read back by spreadsheet and database programs such as Excel.

Monitors

Monitors are another method of displaying information in a model. Here are the monitors in Wolf Sheep Predation:



The monitor labeled time-ticks tells us how much time has passed in the model. The other monitors show us the population of sheep and wolves, and the amount of grass. (Remember, the amount of grass is divided by four to keep the graph from getting too tall.)

The numbers displayed in the monitors update continuously as the model runs, whereas the plots show you data from the whole course of the model run.

Note that NetLogo has also another kind of monitor, called agent monitors. These will be introduced in Tutorial #2.

Controlling the Graphics Window

If you look at the graphics window, youll see a strip of controls along the top edge. The control strip lets you control various aspects of the graphics window.

Let's experiment with the effect of these controls.

- Press setup and then go to start the model running.
- As the model runs, move the slider in the control strip back and forth.

What happens?

This slider is helpful if a model is running too fast for you to see what's going on in detail.

- Move the speed slider all the way to the right again.
- Now try pressing and unpressing the red arrowhead in the control strip.
- Also try pressing and unpressing the on/off switch in the control strip.

What happens?

The shapes button and the display button are useful if you're impatient and want a model to run faster. When shapes are turned off, turtles are drawn as solid squares; it takes less work for NetLogo to draw squares than special shapes, so the model runs faster.

The display button freezes the display. The model continues to run in the background, and plots and monitors still update; but if you want to see what's happening in the graphics window, you need to unfreeze the display by turning the switch back on. Most models run much faster when the display is frozen.

The size of the Graphics Window is determined by three separate settings: Screen Edge X, Screen Edge Y, and Patch Size. Lets take a look at what happens when we change the size of the Graphics Window in the Wolf Sheep Predation model.

- Experiment with the three sets of black arrows on the left of the control strip.

What happens the first time you press one of them?

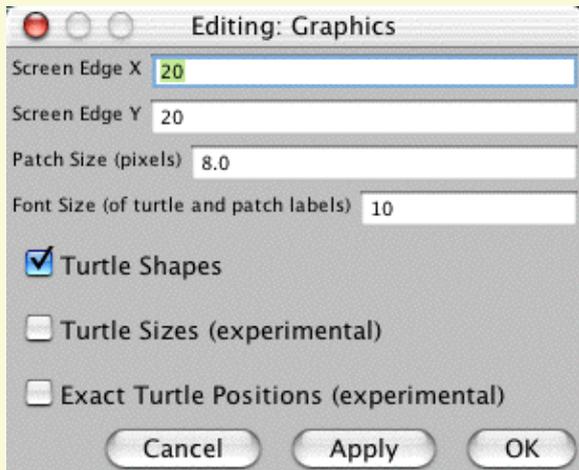
What happens after that? Try all three sets of arrows.

The arrows give you a convenient way of changing the number of patches in the world. NetLogo can't change the number of patches without starting the model over from the beginning, so that's why it warns you the first time you press an arrow.

There are more graphics window settings than there's room for in the control strip. The More... button lets you get to the rest of the settings.

- Press the More... button in the control strip.

A dialog box will open containing all the settings for the Graphics Window:



What are the current settings for Screen Edge X, Screen Edge Y, and Patch Size?

- Press cancel to make this window go away without changing the settings.
- Place your mouse pointer next to, but still outside of, the Graphics Window.

You will notice that the pointer turns into a crosshair.

- Hold down the mouse button and drag the crosshair over the Graphics Window.

The Graphics Window is now selected, which you know because it is now surrounded by a gray border.

- Drag one of the square black handles. The handles are found on the edges and at the corners of the Graphics Window.
- Unselect the graphics window by clicking anywhere in the white background of the Interface tab.
- Press the More... button again and look at the settings.

What numbers changed?

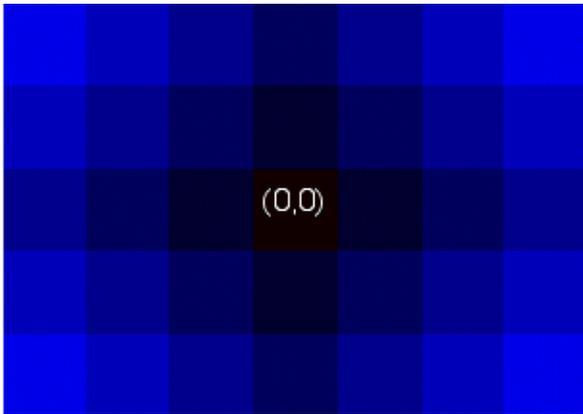
What numbers didnt change?

The NetLogo world is a two dimensional grid of patches. Patches are the individual squares in the grid.

In Wolf Sheep Predation, when the grass? switch is on the individual patches are easily seen, because some of them are green, while others are brown.

Think of the patches as being like square tiles in a room with a tile floor. Exactly in the middle of the room is a tile labeled (0,0); meaning that if the room was divided in half one way and then the other way, these two dividing lines would intersect on this tile. We now have a coordinate system that will

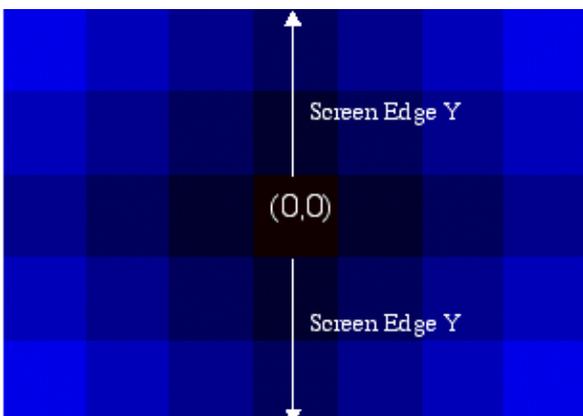
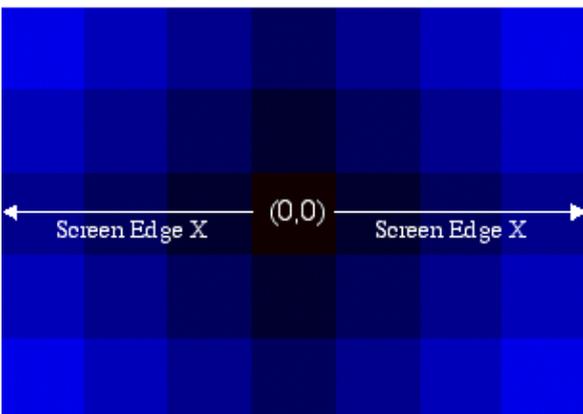
help us locate objects within the room:



How many tiles away is the (0,0) tile from the right side of the room?

How many tiles away is the (0,0) tile from the left side of the room?

In NetLogo, the distance the middle tile is from the right or left edge of the room this is called Screen Edge X. And the distance the middle tile is from the top and bottom edges is called Screen Edge Y:



In these diagrams, Screen Edge X is 3 and Screen Edge Y is 2.

When you change the patch size, the number of patches (tiles) doesn't change, the patches only get larger or smaller on the screen.

Let's look at the effect of changing Screen Edge X and Screen Edge Y.

- Using the Edit dialog that is still open, change Screen Edge X to 30 and Screen Edge Y value to 10.

What happened to the shape of the Graphics Window?

- Press the setup button.

Now you can see the new patches you have created.

- Edit the Graphics Window again.
- Change the patch size to 20 and press OK.

What happened to the size of the Graphics Window? Did its shape change?

Editing the Graphics window also lets you change two other settings: the font size of labels, and whether the Graphics Window uses shapes. Feel free to experiment with these settings as well.

Once you are done exploring the Wolf Sheep Predation model, you may want to take some time just to explore some of the other models available in the Models Library.

The Models Library

The library contains five sections: Sample Models, Curricular Models, Code Examples, HubNet Calculator Activities, HubNet Computer Activities.

Sample Models

The Sample Models section is organized by subject area and currently contains more than 130 models. We are continuously working on adding new models to it, so come visit this section at a later date to view the new additions to the library.

Some of the folders in Sample Models have subfolders labeled (unverified). These models are still in the process of being tested and reviewed for content and accuracy. They are complete and functional; we just haven't tested them as thoroughly, or made sure the code is good sample code.

Curricular Models

These are models designed to be used in schools in the context of curricula developed by the CCL at Northwestern University. Some of these are models are also listed under Sample Models; others are unique to this section. See the info tabs of the models for more information on the curricula they go with.

Code Examples

These are simple demonstrations of particular features of NetLogo. They'll be useful to you later when you're extending existing models or building new ones. For example, if you wanted to put a histogram within your model, you'd look at Histogram Example to find out how.

HubNet Calculator & Computer Activities

This section contains participatory simulations for use in the classroom. For more information about HubNet, see the [HubNet Guide](#).

Whats Next?

If you want to learn how to explore models at a deeper level, [Tutorial #2: Commands](#) will introduce you to the NetLogo modeling language.

In [Tutorial #3: Procedures](#) you can learn how to alter and extend existing models and build your own models.

Tutorial #2: Commands

In Tutorial #1, you had the opportunity to view some of the NetLogo models, and you have successfully navigated your way through opening and running models, pressing buttons, changing slider and switch values, and gathering information from a model using plots and monitors. In this section, the focus will start to shift from observing models to manipulating models. You will start to see the inner workings of the models and be able to change how they look.

Sample Model: Traffic Basic

- Go to the Models Library (File menu).
- Open up Traffic Basic, found in the Social Science section.
- Run the model for a couple minutes to get a feel for it.
- Consult the Information tab for any questions you may have about this model.

In this model, you will notice one red car in a stream of blue cars. The stream of cars are all moving in the same direction. Every so often they pile up and stop moving. This is modeling how traffic jams can form without any cause such as an accident, a broken bridge, or an overturned truck. No centralized cause is needed for a traffic jam to form.

You may alter the settings and observe a few runs to get a full understanding of the model.

As you are using the Traffic Basic model, have you noticed any additions you would like to make to the model?

Looking at the Traffic Basic model, you may notice the environment is fairly simple; a black background with a white street and number of blue cars and one red car. Changes that could be made to the model include: changing the color and shape of the cars, adding a house or street light, creating a stop light, or even creating another lane of traffic. Some of these suggested changes are cosmetic and would enhance the look of the model while the others are more behavioral. We will be focusing more on the simpler or cosmetic changes throughout most of this tutorial. ([Tutorial #3](#) will go into greater detail about behavioral changes, which require changing the Procedures tab.)

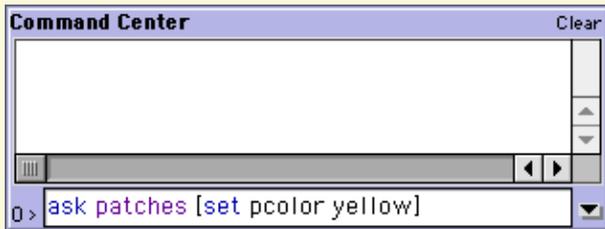
To make these simple changes we will be using the Command Center.

The Command Center

The Command Center is located in the Interface Tab and allows you to enter commands or directions to the model. Commands are instructions you can give to NetLogos agents: turtles, patches, and the observer. (Refer to the [Interface Guide](#) for details explaining the different parts of the Command Center.)

In Traffic Basic:

- Press the setup button.
- Locate the Command Center.
- Click the mouse in the white box at the bottom of the Command Center.
- Type the text shown here:



- Press the return key.

What happened to the Graphics Window?

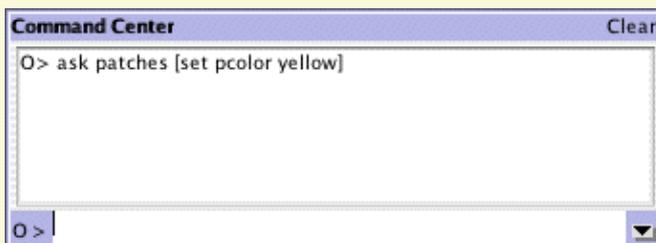
You may have noticed the background of the Graphics Window has turned all yellow and the street has disappeared.

Why didnt the cars turn yellow too?

Looking back at the command that was written, we asked only the patches to change their color. In this model, the cars are represented by a different kind of agent, called turtles. Therefore, the cars did not received these instructions and thus did not change.

What happened in the Command Center?

You may have noticed that the command you just typed is now displayed in the white box in the middle of the Command Center as shown below:

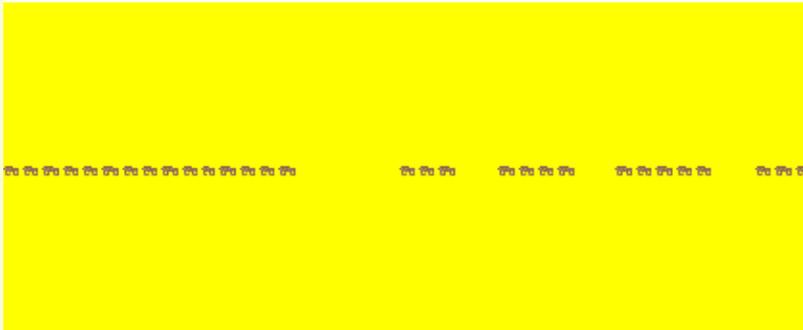


- Type in the white box at the bottom of the Command Center the text shown below:



Was the result what you expected?

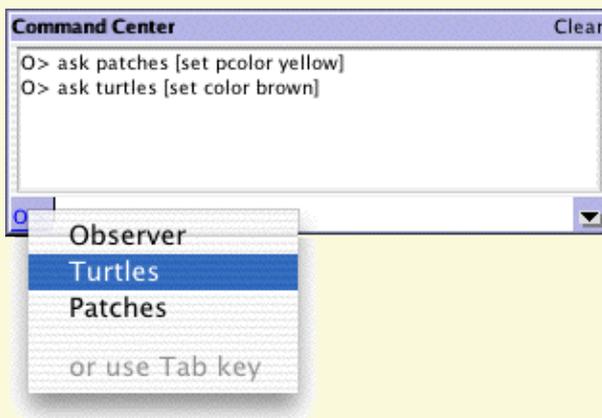
Your Graphics Window should have a yellow background with a line of brown cars in the middle of the window:



The NetLogo world is a two dimensional world that is made up of turtles, patches and an observer. The patches create the ground in which the turtles can move around on and the observer is a being that oversee everything that is going on in the world. (For a detailed description and specifics about this world, refer to the [NetLogo Programming Guide](#).)

In the Command Center, we have the ability to give the observer a command, the turtles a command, or the patches a command. We choose between these options by using the popup menu located in the bottom left corner of the Command Center. You can also use the tab key on your keyboard to cycle through the different options.

- In the Command Center, click on the O> in the bottom left corner:



- Choose Turtles from the popup menu.
- Type `set color pink` and press return.
- Press the tab key until you see P> in the bottom left corner.
- Type `set pcolor white` and press return.

What does the Graphics Window look like now?

Do you notice any differences between these two commands and the observer commands from earlier?

The observer oversees the world and therefore can give a command to the patches or turtles using `ask`. Like in the first example (`O>ask patches [set pcolor yellow]`), the observer has to ask the patches to set their `pcolor` to yellow. But when a command is directly given to a group of agents like in the second example (`P>set pcolor white`), you only have to give the command itself.

- Press setup.

What happened?

Why did the Graphic Window revert back to the old version, with the black background and white road? Upon pressing the setup button, the model will reconfigure itself back to the settings outlined in the Procedures tab. The Command Center is not often used to permanently change the model. It is most often used as a tool to customize current models and allows for you to manipulate the NetLogo world to further answer those What if questions that pop up as you are investigating the models. (The Procedures tab is explained in the next tutorial, and in the [Programming Guide](#).)

Now that we have familiarized ourselves with the Command Center, lets look at some more details about how colors work in NetLogo.

Working With Colors

You may have noticed in the previous section that we used two different words for changing color: `color` and `pcolor`.

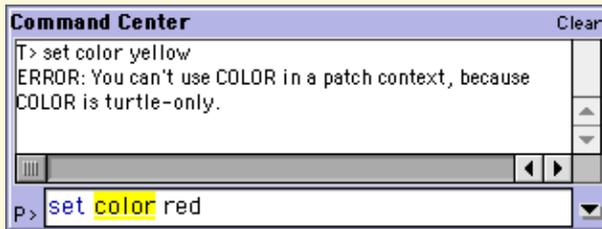
What is the difference between `color` and `pcolor`?

- Choose Turtles from the popup menu in the Command Center (or use the tab key).
- Type `set color blue` and press return.

What happened to the cars?

Think about what you did to make the cars turn blue, and try to make the patches turn red.

If you try to ask the patches to `set color red`, an error message occurs:



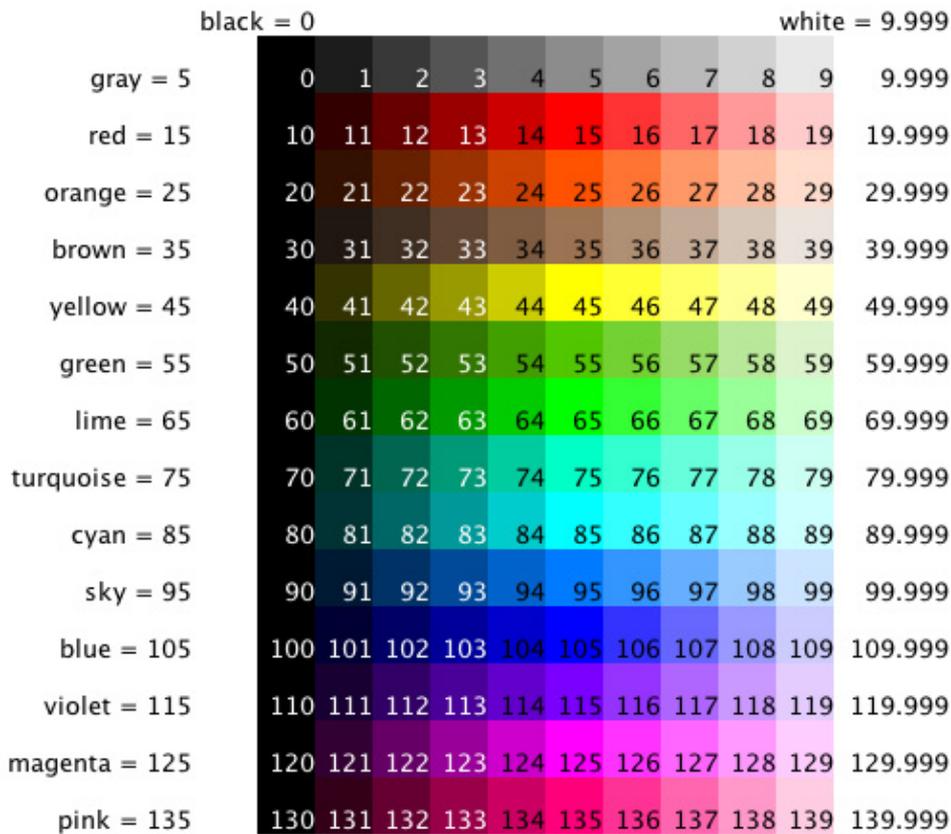
- Type `set pcolor red` instead and press return.

We call `color` and `pcolor` variables. Some commands and variables are specific to turtles and some are specific to patches. For example, the `color` variable is a turtle variable, while the `pcolor` variable is a patch variable.

Go ahead and practice altering the colors of the turtles and patches using the `set` command and these two variables.

To be able to make more changes to the colors of turtles and patches, or shall we say cars and backgrounds, we need to gain a little insight into how NetLogo deals with colors.

In NetLogo, all colors have a numeric value. In all of the exercises we have been using the name of the color. This is because NetLogo recognizes 16 different color names. This does not mean that NetLogo only recognizes 16 colors. There are many shades in between these colors that can be used too. Heres a chart that shows the whole NetLogo color space:



To get a color that doesn't have its own name, you just refer to it by a number instead, or by adding or subtracting a number from a name. For example, when you type `set color red`, this does the same thing as if you had typed `set color 15`. And you can get a lighter or darker version of the same color by using a number that is a little larger or a little smaller, as follows.

- Choose Patches from the popup menu in the Command Center (or use the tab key).
- Type `set pcolor red - 2` (The spacing around the `-` is important.)

By subtracting from red, you make it darker.

- Type `set pcolor red + 2`

By adding to red, you make it lighter.

You can use this technique on any of the colors listed in the chart.

Agent Monitors and Agent Commanders

In the previous activity, we used the `set` command to change the colors of all the cars. But if you recall, the original model contained one red car amongst a group of blue cars. Let's look at how to change only one car's color.

- Press setup to get the red car to reappear.
- If you are on a Macintosh, hold down the Control key and click on the red car. On other operating systems, click on the red car with the right mouse button.
- From the popup menu that appears, choose inspect turtle 0

A turtle monitor for that car will appear:

turtle 0	
who	0
color	15.0
heading	90.0
xcor	-13.0
ycor	0.0
shape	"car"
pen-down?	false
label	
label-color	9.9999
breed	turtles
hidden?	false
size	1.0
speed	0.6
speed-limit	1
speed-min	0

Taking a closer look at this turtle monitor, we can see all of the variables that belong to the red car. A variable is a place that holds a value that can be changed. Remember when it was mentioned that all colors are represented in the computer as numbers? The same is true for the agents. For example, turtles have an ID number we call their who number.

Lets take a closer look at the turtle monitor:

What is this turtles who number?

What color is this turtle?

What shape is this turtle?

This turtle monitor is showing a turtle who that has a who number of 0, a color of 15 (red see above chart), and the shape of a car.

There are two other ways to open a turtle monitor besides right-clicking (or control-clicking, depending on your operating system). One way is to choose Turtle Monitor from the Tools menu, then type the who number of the turtle you want to inspect into the who field and press return. The other way is to type `inspect turtle 0` (or other who number) into the Command Center.

You close a turtle monitor by clicking the close box in the upper left hand corner (Macintosh) or upper right hand corner (other operating systems).

Now that we know more about Agent Monitors, we have three ways to change an individual turtles color.

One way is to use the box called an Agent Commander found at the bottom of an Agent Monitor. You type commands here, just like in the Command Center, but the commands you type here are only done by this particular turtle.

- In the Agent Commander of the Turtle Monitor for turtle 0, type `set color pink`.

What happens in the Graphics Window?

Did anything change in the Turtle Monitor?

A second way to change one turtles color is to go directly to the color variable in the Turtle Monitor and change the value.

- Select the text to the right of color in the Turtle Monitor.
- Type in a new color such as `green + 2`.

What happened?

The third way to change an individual turtles or patches color is to use the observer. Since, the observer oversees the NetLogo world, it can give commands that affect individual turtles, as well as groups of turtles.

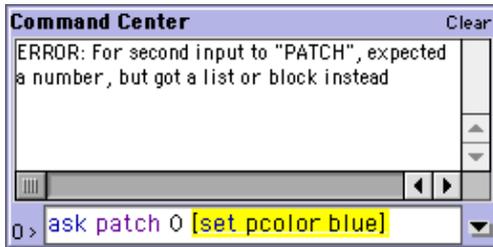
- In the Command Center, select Observer from the popup menu (or use the tab key).
- Type `ask turtle 0 [set color blue]` and press return.

What happens?

Just as there are Turtle Monitors, there are also Patch Monitors. Patch monitors work very similarly to Turtle Monitors.

Can you make a patch monitor and use it to change the color of a single patch?

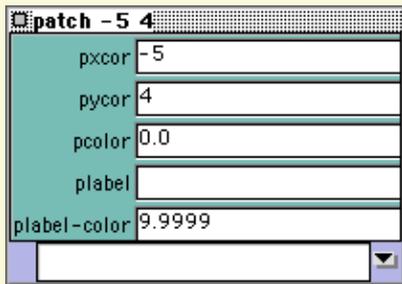
If you try to have the observer `ask patch 0 [set pcolor blue]`, youll get an error message:



To ask an individual turtle to do something, we use its who number. But patches don't have who numbers, therefore we need to refer to them some other way.

Remember, patches are arranged on a coordinate system. Two numbers are needed to plot a point on a graph: an x-axis value and a y-axis value. Patch locations are designated in the same way as plotting a point.

- Open a patch monitor for any patch.



The monitor shows that for the patch in the picture, its `pxcor` variable is -5 and its `pycor` variable is 4. If we go back to the analogy of the coordinate plane and wanted to plot this point, the point would be found in the upper left quadrant of the coordinate plane where $x=-5$ and $y=4$.

To tell this particular patch to change color, use its coordinates.

- Type `ask patch -5 4 [set pcolor blue]` and press return.

What are the two words in this command that tip you off that we are addressing a patch?

Whats Next?

At this point, you may want to take some time to try out the techniques you've learned on some of the other models in the Models Library.

In [Tutorial #3: Procedures](#) you can learn how to alter and extend existing models and build your own models.

Tutorial #3: Procedures

In Tutorial #2, you learned how to use command centers and agent monitors to inspect and modify agents and make them do things. Now you're ready to learn about the real heart of a NetLogo Model: the Procedures tab. This tutorial leads you through the process of building a complete model, built up stage by stage, with every step explained along the way.

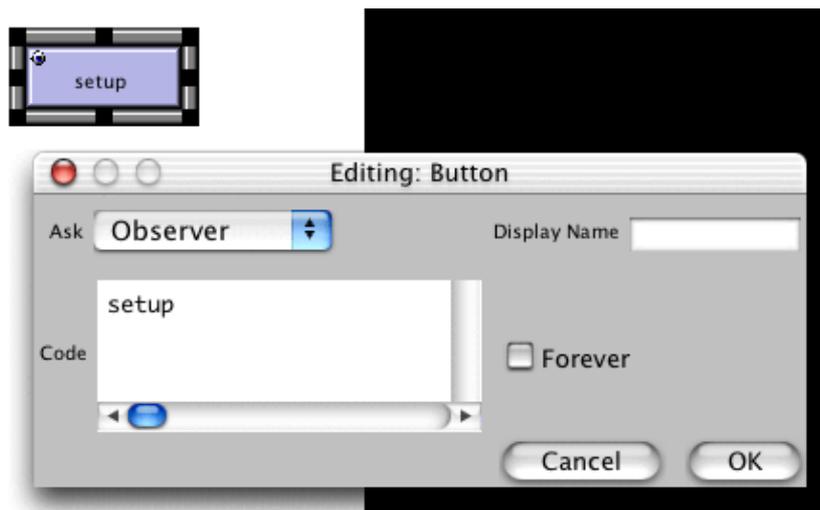
You've already been exposed to the three types of agents you can give commands to in NetLogo: turtles, patches, and the observer. As you start to write your own procedures, it'll be helpful to keep in mind how people usually think of these three different kinds of agents. The turtles and patches usually don't use information about the whole world. They mostly use information about what's close to them. The observer, on the other hand, typically uses and accesses the whole world. Also, while patches can't move and often represent some sort of environment, turtles can move around in the world.

Setup and Go

To start a new model, select New from the the File menu. Then begin making your model by creating a once-button called setup.

Here's how to make the button:

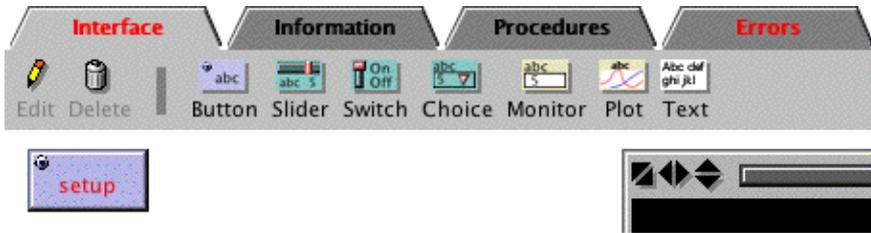
1. Click on the button icon in the Toolbar
2. Click where you want the button to be in the empty white area of the Interface tab
3. When the dialog box for editing the properties of the button opens, type `setup` in the box labeled Code



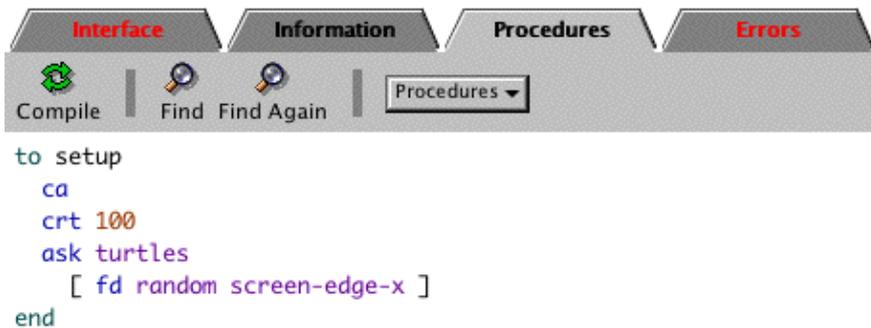
4. Press OK to dismiss the dialog box

Now you have a button called setup. It will execute the procedure setup when pressed, which once we define it, will do just that set up the NetLogo world.

At this point, the Errors tab has turned red. That's because there is no procedure called setup! If you want to see the actual error message, switch to the Errors tab:



Now switch to the Procedures Tab and create the setup procedure shown below. Notice that the lines are indented different amounts. A lot of people find it very helpful to indent their code in a way at least similar to how its done here. It helps them keep track of where theyre at inside of a procedure and makes what they write easier for others to read as well.



One line at a time:

to setup begins defining a procedure named setup.

ca is short for **clear-all** (you can also spell it out if you want). This command will blank out the screen, initialize any variables you might have to 0, and remove all turtles. Basically, it wipes the slate clean for a new run of the project.

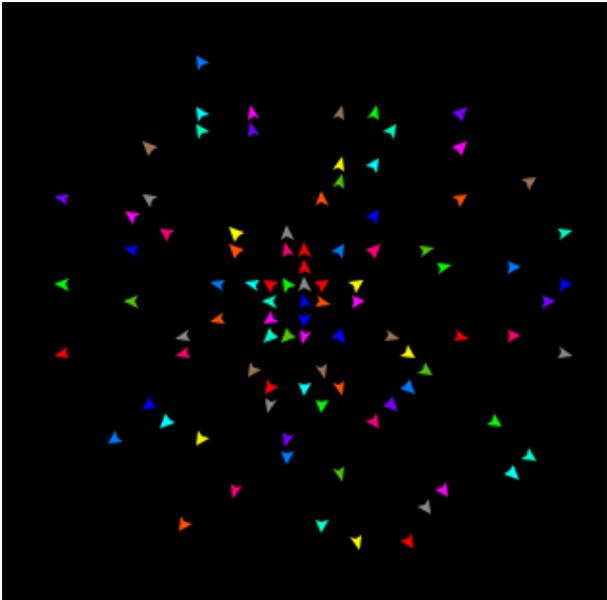
crt 100 will then create 100 turtles. If the turtles didnt move after this command is given, each of these turtles would begin on the center patch (at location 0,0). You would only see what looks like one turtle on the screen; theyd all be on top of each other lots of turtles can share the same patch. Only the last turtle to arrive on the patch would be visible. Each of these newly-created turtles has its own color, its own heading. All of the turtles are evenly distributed around the circle.

ask turtles [...] tells each turtle to execute, independently, the instructions inside the brackets. Note that **crt** is not in brackets. If the agent (observer, turtle, or patch) is not specified, the observer executes it.

fd (random screen-edge-x) is a pair of commands. Each turtle will first evaluate the expression **random screen-edge-x** which will return a random value between 0 and screen-edge-x (the dimension from the center to the edge of the screen along the x-axis, maybe 20 or so). It then takes this number, and goes **fd** (short for **forward**) that number of steps, in the direction of its heading.

end completes the definition of the setup procedure.

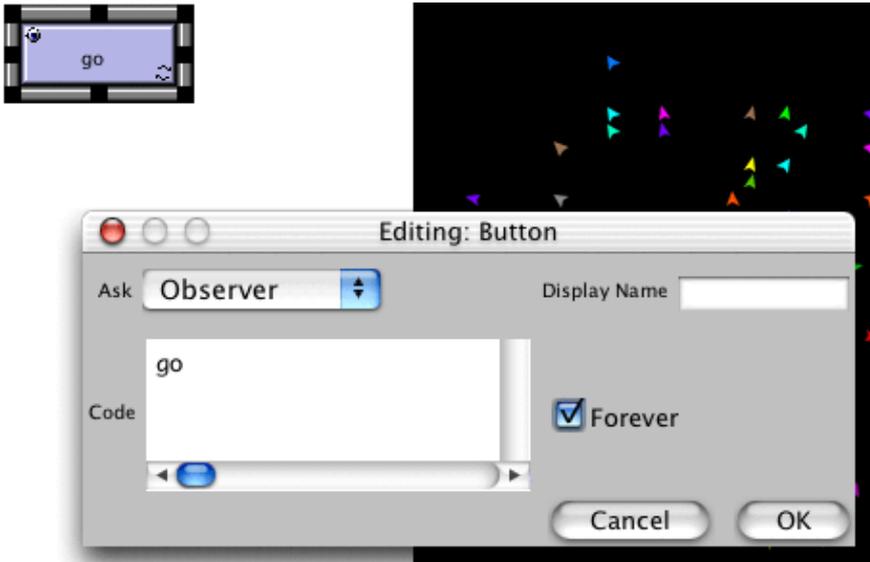
Press your setup button when youre done writing the code. You will see the turtles quickly spread out in a rough cluster:



Notice the density distribution of the turtles on the Graphics Window. Press setup a couple more times, and watch how the turtles arrangement changes. Keep in mind that some of the patches may have multiple turtles on them.

Can you think of other ways to randomly distribute the turtles over the screen? Note that if a turtle moves off the screen, it wraps, that is, comes in the other side.

Make a forever-button called go. Again, begin by creating a button, but this time check the forever checkbox in the edit window.



Then write its procedure:

```
to go
  move-turtles
end
```

But what is **move-turtles**? Is it a primitive (a built-in function of NetLogo), like **fd** is? No, its a procedure that youre about to write, right after the **go** procedure:

```
to move-turtles
  ask turtles [
    set heading (random 360)
    fd 1
  ]
end
```

Be careful of the spacing around the **-**. In Tutorial #2 we wanted `red - 2` but now we want `move-turtles`. The **-** combines `move` and `turtles` into one word.

Line by line:

ask turtles [commands] says that each turtle should execute the commands in the brackets.

set heading (random 360) is another two-step command. First, each turtle picks a random integer between 0 and 359 (**random** doesnt include the number you give it). Then the turtle sets its heading to the number the turtle picked. Heading is measured in degrees, clockwise around the circle, starting with 0 degrees at twelve oclock.

fd 1: Each turtle moves forward one step in the new direction it just set its heading to.

Why couldnt we have just written that in **go**? We could, but during the course of building your project, its likely that youll add many other parts. Wed like to keep **go** as simple as possible, so that it is easy to understand. Eventually, it could include many other things you want to have happen as the model runs, such as calculating something or plotting the results. Each of these sub-procedures could have its own name.

The **go** button is a forever-button, meaning that it will continually execute its code until you shut it off (by clicking on it again). After you have pressed **setup** once, to create the turtles, press the **go** button. Watch what happens. Turn it off, and youll see that all turtles stop in their tracks.

We suggest you start experimenting with other turtle commands. You might try typing **T> pendown** into the Command Center and then pressing **go**. Another thing to try is changing **set heading (random 360)** to **rt (random 360)** inside of **move-turtles**. (**rt** is short for right turn.) Also, you can try changing **set heading (random 360)** to **lt (random 45)** inside of **move-turtles**. Type commands into the Command Window (like **set colorred**), or add them to **setup**, **go**, or **move-turtles**. Note that when you enter commands in the Command Center, you must choose **T>**, **P>**, or **O>** in the popup menu on the left, depending on which agents are going to execute the commands. You can also use the tab key, which you might find more convenient than using the popup menu. **T>commands** is identical to **O> ask turtles [commands]**, and **P>commands** is identical to **O> ask patches [commands]**.

Play around. Its easy and the results are immediate and visible one of NetLogos many strengths. Regardless, the tutorial project continues...

Patches and Variables

Now weve got 100 turtles aimlessly moving around, completely unaware of anything else around

them. Lets make things a little more interesting by giving these turtles a nice background against which to move. Go back to the setup procedure. We can rewrite it as follows:

```
patches-own [elevation]

to setup
  ca
  setup-patches
  setup-turtles
end
```

The line at the top, **patches-own [elevation]** declares that we have a variable for the patches, called **elevation**. Our **setup-patches** procedure that we havent defined yet will then use this variable. We also still need to define **setup-turtles** as well, but, for now, heres how to define **setup-patches**:

```
to setup-patches
  ask patches
    [ set elevation (random 10000) ]
  diffuse elevation 1
  ask patches
    [ set pcolor scale-color green elevation 1000 9000 ]
end
```

The **setup-patches** procedure gives every patch an elevation and a color. First, each patch picks a random number between 0 and 9999, inclusive, and sets its *elevation* value to that number.

We then use an observer primitive, **diffuse**, that essentially smoothes out the distribution of this variable over the neighboring patches. Remember that primitives are built in functions of NetLogo, as opposed to functions (or procedures) that you define.

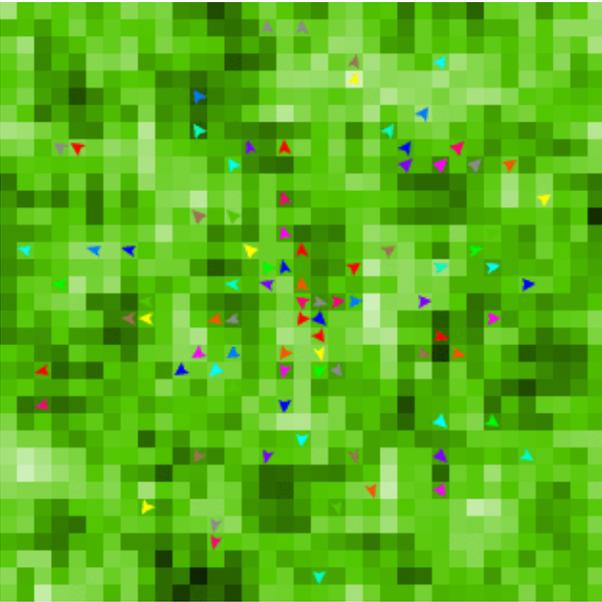
Scale-color is a reporter that uses the different values of *elevation* to assign colors to the patches. In this case, were assigning different shades of green to all the patches. (Dont worry about the numbers given to **diffuse** and **scale-color** just yet...) The larger *elevation* is, the lighter the shade of green. Low values of *elevation* will result in darker shades.

The only part remaining in our new setup that is still undefined is **setup-turtles**:

```
to setup-turtles
  crt 100
  ask turtles
    [ fd (random screen-edge-x) ]
end
```

Setup-turtles is exactly what we were doing in the old **setup** procedure.

After typing all of this in, press the setup button back in the models interface. Voila! A lush NetLogo landscape complete with turtles and patches appears. After seeing the new setup work a few times, you may find it helpful to read through the procedure definitions again.



Here's a way for you to see what **diffuse** does. Return to the Procedures Window, and use a semicolon to deactivate the diffuse command like this:

```
;diffuse elevation 1
```

Semicolons are very useful in writing procedures. They can be used as above to save you from having to delete commands to try something else out and then having to rewrite them. Also, they can be used to add some explanatory text to your procedures. A lot of people like to do this to make their procedures more readable to others. Notice that all the text to the right of a semicolon becomes gray.

Press setup again. It looks different, doesn't it? This is because, as mentioned above, **diffuse** has each patch share its value of *elevation* with all its neighbors, by having every patch reset its value of *elevation* to a new value that depends on the value of *elevation* all around it. For further explanation of how diffuse works, go to the [Primitives Dictionary](#) if you'd like. Also, it may help to toy with the values being passed to it and see what happens.

We're now prepared to create some kind of dialog between the turtles and the patches. In fact, we even have an idea for a project here. Notice that we called the patch variable *elevation*, and that our landscape sort of looks topographical? We're going to have our turtles do what is called hill-climbing, where every turtle seeks to find the highest elevation it can.

In order to do this, we will use compound commands—a feature that makes NetLogo quite different from StarLogoT (and somewhat closer to natural language). Go to the Command Center, and type **O> show max values—from patches [*elevation*]** and **show min values—from patches [*elevation*]**. These two reporters will, respectively, search over all the patches to return to you the highest elevation and the lowest. These commands work like this (you can read about them in the NetLogo [Primitives Dictionary](#)):

Look up values—from in the dictionary. It shows values—from AGENTSET [expression] and says it returns a list. In this case, it looks at the expression (*elevation*) for each agent in the agentset (patches) and returns all of these as a list of elevations.

Look up `min` in the dictionary. It shows `min list` and says its a reporter. So it takes the list of elevations and reports the smallest value.

`Show` displays this value in the command center.

We will use these compound statements **`max values-from patches [elevation]`** and **`min values-from patches [elevation]`** in our model.

Just in case we end up needing the highest and lowest elevation in several places in our procedures, lets make a shortcut. Well do a little extra work now so that if we need these values later, well have a shortcut to use. First, at the top of your code (right after the `patches-own` declaration), declare two global variables as such:

```
globals [highest ;; the highest patch elevation
         lowest] ;; the lowest patch elevation
```

(Notice the use of semicolons here. Although the names of the global variables are descriptive, the semicolons allow us to describe the variables even more.)

Global variables can be used by all the agents in the model. In particular, patches can use `highest` and `lowest` in the **`setup-patches`** procedure. We need to store the highest and lowest elevations in these global variables once, and then everyone will have quick access to them after that. Write:

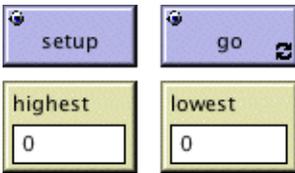
```
to setup-patches
  ask patches
    [ set elevation (random 10000) ]
  diffuse elevation 1
  ask patches
    [ set pcolor scale-color green elevation 1000 9000 ]
  set highest max values-from patches [elevation]
  set lowest min values-from patches [elevation]
  ask patches [
    if (elevation > (highest - 100))
      [set pcolor white]
    if (elevation <(lowest + 100))
      [set pcolor black] ]
end
```

Now we have saved the highest and lowest points in our terrain and displayed them graphically.

Look at the last two statements, the **`if`**-statements. Each patch looks at both of these statements, and compares its own value of *elevation* to our global variables *highest* and *lowest*. If the comparison evaluates to true, the patch executes the list of commands inside the brackets. In this case, the patch changes its color. If the comparison evaluates to false, the patch skips over the commands inside the brackets.

These **`if`**-statements have as a result that all patches whose value of *elevation* is NEAR to the highest (within about 1% for our values) change their color to white, and all patches whose values are NEAR to the lowest become black. We want this so that theyll be easier to see. You can make a couple of quick changes here if you wish they wont affect the rest of the model. For example, instead of saying `set pcolor white` and `set pcolor black`, you can say `set pcolor blue` and `set pcolor red` (or whatever other colors you may wish). Also, you can change the range of highest peaks and lowest peaks by changing the number 100 to some other number.

After this, create two monitors in the Interface Window with the Toolbar. (You make them just like buttons and sliders, using the monitor icon on the Toolbar.) Name one of them `highest` and the other one `lowest`. The reporters you'll want in each of them happen to be `highest` and `lowest` as well. (If you want to learn more about reporters, you can look them up in the [NetLogo Programming Guide](#).) Now every time you click `setup` and redistribute the values of `elevation`, you'll know exactly what the highest and lowest elevations are, and where they can be found.



An Uphill Algorithm

Okay. Finally we're ready to start hill-climbing. To rehash: we've got some turtles randomly spread out from the origin; and we've got a landscape of patches, whose primary attribute is their *elevation*. Lastly, we have two kinds of tools to help us understand the patch landscape: each patch has a color, depending on its value of *elevation*, and we have a pair of monitors telling us what the highest peak and lowest valley are. What we need now is for the turtles to wander around, each trying to get to the patch that has the highest elevation.

Let's try a simple algorithm first. We'll assume three things: 1), that the turtles cannot see ahead farther than just one patch; 2), that each turtle can move only one square each turn; and 3), that turtles are blissfully ignorant of each other. Before, we had a procedure *move-turtles* like this:

```
to move-turtles
  ask turtles [
    set heading (random 360)
    fd 1
  ]
end
```

But now we don't want them to move randomly about. We want each turtle to look at the *elevation* of each patch directly around it, and move to the patch with the highest elevation. If none of the patches around it have a higher elevation than the patch it is on, it'll stay put. This new procedure should replace *move-turtles* inside of *go*. Type in the following code and run it once or twice:

```
;; each turtle goes to the highest elevation in a radius of one
to move-to-local-max
  ask turtles [
    set heading uphill elevation
    if ( elevation-of patch-at dx dy > elevation )
      [ fd 1 ]
  ]
end
```

Now that you've seen the uphill algorithm work in the model, let's go through the new commands involved. (If you haven't run the model yet since writing *move-to-local-max*, give it a try.) There are five new commands here: *uphill*, *elevation-of*, *patch-at*, and *dx* and *dy*. *uphill elevation* finds the heading to the patch with the highest value of *elevation* in the patches in a one-patch radius of the turtle. Then through the use of the command *set heading*, the turtle sets its heading to that direction. *elevation-of patch-at dx dy* has each turtle look at the variable *elevation* in the patch on which the

turtle would be if it went forward 1. If the test is true, the turtle moves itself forward 1. (The test is necessary because if the turtle is already on the peak, we dont want it to move off it!)

Go ahead and type that in, but before you test it out by pressing the go button, ask yourself this question: what do you think will happen? Try and predict how a turtle will move, where it will go, and how long itll take to get there. When youre all set, press the button and see for yourself.

Surprised? Try to understand why the turtles converge to their peaks so quickly. Maybe you dont believe the algorithm weve chosen works correctly. Theres a simple procedure you can make to test it. write a procedure **recolor-patches** so that it says:

```
to recolor-patches
  ask patches
  [
    set elevation pycor
    set pcolor scale-color green elevation
                          (0 - screen-edge-y) screen-edge-y
  ]
end
```

Press setup. The model looks the same as it did before because **recolor-patches** hasnt been run yet. Instead of making a button that calls your testing procedure, lets do something different. Type O>recolor-patches into the command center, the procedure gets called. Now, when you press go, see that the turtles all head for the highest elevation the top of the screen.

Another common tool to see whats going on is to write T> pd in the Command Center. Then each turtle traces its path with its color. This will show you where the turtle has been.

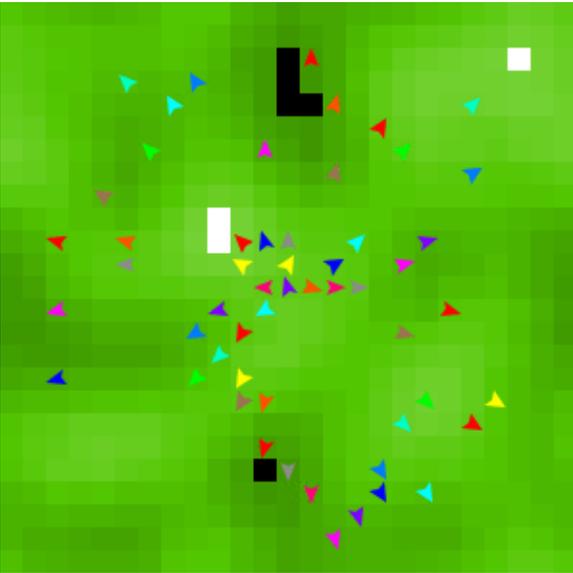
Our turtles rapidly arrive at local maxima in our landscape. Local maxima and minima abound in a randomly generated landscape like this one. Our goal is to still get the turtles to find an optimal maximum, which is one of the white patches.

Part of the problem is that our terrain is terribly lumpy. Every patch picked a random elevation, and then we diffused these values one time. This really doesnt give us a continuous spread of elevation across the graphics window, as you might have noticed. We can correct this problem to an arbitrary degree by diffusing more often. Replace the line:

```
diffuse elevation 1
```

with:

```
repeat 5 [ diffuse elevation 1 ]
```



The **repeat** primitive is another way for NetLogo to loop besides the forever-button. **Repeat** takes a number (here, 5) and a command list (here, the **diffuse** command), and executes the command list that number of times (here, five times). Try it out, and look at the landscape (i.e. press setup and see what you think). Then, press go and watch the turtles performance. (Remember that the lighter the patch, the greater the elevation.)

Obviously, fewer peaks make for an improvement in the turtles performance. On the other hand, maybe you feel like this is cheating the turtles really aren't doing any better, its just that their problem was made easier. True enough. If you call **repeat** with an even higher number (20 or so), you'll end up with only a handful of peaks, as the values become more evenly distributed with every successive call. Watch your monitor values.

In order to specify how smooth you want your world to be, lets make it easier to try different values. Maybe one time you'll want the turtles to try and solve a hard world, and maybe another time you'll just want to look at an easy landscape. So we'll make a global variable named smoothness. Create a slider in the Interface Window and call it smoothness in the editing box. The minimum can be 0, and the maximum can be 25 or so. Then change your code to:

```
repeat smoothness [ diffuse elevation 1 ]
```

Experiment with the turtles performance in different terrains.

We still haven't even begun to solve the problem of getting all the turtles to the highest elevation, though. So far we've just been getting the turtles to the highest point that's near them. If a turtle starts off in one corner of the world on a hill and there's a mountain in a different corner, the turtle will never find the mountain. To find the mountain, the turtle would have to go down off the hill first, but in our model, turtles only move up. Notice that the individual turtles don't use highest anywhere. The turtles just look at elevations close to them and go the highest point they can see.

Before trying something else, it'd be nice if we could have some other, more precise method for evaluating the turtles performance. Fortunately, NetLogo allows us to plot data as we go along.

To make plotting work, we'll need to create a plot in the Interface tab, and set some settings in it. Then we'll add one more procedure to the Procedures tab, which will update the plot for us.

Lets do the Procedures tab part first. Change **go** to call the new procedure were about to add:

```
to go
  move-to-local-max
  do-plots
end
```

Now add the new procedure. What were plotting is the number of turtles whove reached our peak-zone (within 1% of the highest elevation) at some given time.

```
to do-plots
  set-current-plot "Turtles at Peaks"
  plot count turtles with
    [ elevation >= (highest - 100) ]
end
```

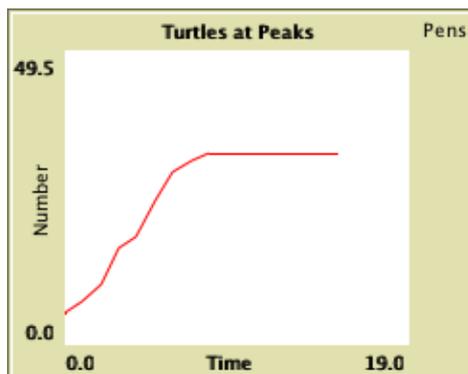
Note that we use the **plot** primitive to add the next point to a plot, but before doing that, we need to tell NetLogo which plot we want, since later our model might have more than one plot.

Thus were plotting the number of turtles within 100 units of our maximum elevation at some given point in time. The **plot** command moves the current plot pen to the point that has x- coordinate equal to 1 greater than the old x- coordinate and y-coordinate equal to the value given in the plot command (in this case, the number of turtles with [elevation >= (highest - 100)]). Then the *plot* command draws a line from the current position of the plot pen to the last point it was on.

In order for `set-current-plot Turtles at Peaks` to work, youll have to add a plot to your model in the Interface tab, then edit it so its name is Turtles at Peaks, the exact same name used in the code. Even one extra space will throw it off it must be exactly the same in both places.

Note that when you create the plot you can set the minimum and maximum values on the x and y axes, and the color of the default plot pen (pick any color you like). Youll want to leave the Autoplot? checkbox checked, so that if anything you plot exceeds the minimum and maximum values for the axes, the axes will automatically grow so you can see the whole plot.

Now reset the project and run it again. You can now watch the plot be created as the model is running. If you notice that your plot doesnt look exactly like the picture below, try to think about why it doesnt look the same. If you think its because go remains pressed until you manually unpress it, well fix that problem by the end of the tutorial. Remember that we kept Autoplot? on. This allows the plot to readjust itself when it runs out of room.



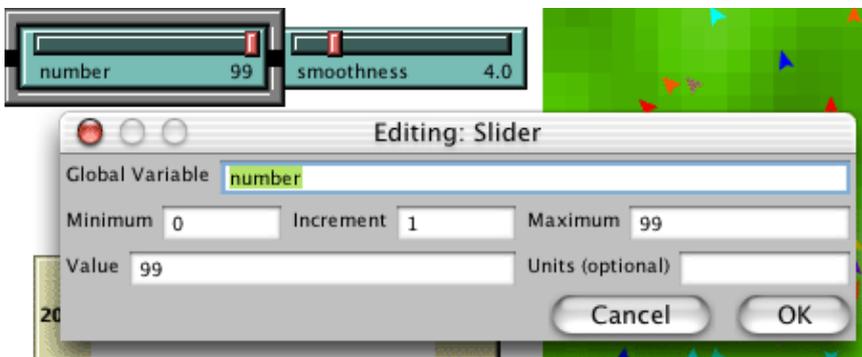
You might try running the model several times under different settings (i.e. different values of *smoothness*) and watch how fast the plot converges to some value, and what fraction of the turtles make it to the top. You may want to even try the same settings several times.

Some More Details

There are a few miscellaneous bugs and quirks you might have already noticed. Here are some quick changes you can make: First, we have a green landscape a naturally green turtle is going to be hard to see. In the ask turtles block in `setup-turtles`, you can say:

```
if (color = green)
  [ set color red ]
```

Second, instead of always using 100 turtles, you can have a variable number of turtles. Make a slider variable (say, `number`):



Then, inside of `setup-turtles`, instead of `crt 100`, you can type:

```
crt number
```

How does using more or fewer turtles affect the success value displayed by the plot?

Third, when all the turtles have found their local maxima, wouldn't it be nice for the model to stop? This requires a few lines of code.

- Add a global variable `turtles-moved?` to the globals list:

```
globals [
  highest           ;; maximum patch elevation
  lowest            ;; minimum patch elevation
  turtles-moved?   ;; so we know when to stop the model
]
```

- At the end of the `go` procedure, add a test to see if any turtles have moved.

```
to go
  set turtles-moved? false
  move-to-local-max
  do-plots
  if (not turtles-moved?)
    [ stop ]
end
```

- In **move-to-local-max** if a turtle moves, set *turtles-moved?* to true.

```

to move-to-local-max
  ask turtles [
    set heading uphill elevation
    if ( elevation-of patch-at dx dy > elevation )
    [
      fd 1
      set turtles-moved? true
    ]
  ]
end

```

Finally, what rules can you think of that would help turtles escape from lower peaks and all get to the highest ones? Try writing them.

Whats Next?

So now you have a nice framework for exploring this problem of hill-climbing, using all sorts of NetLogo modeling features: buttons, sliders, monitors, plots, and the graphics window. Youve even written a quick procedure to give the turtles something to do. And thats where this tutorial leaves off.

If youd like to look at some more documentation about NetLogo, the [Interface Guide](#) section of the manual walks you through every element of the NetLogo interface in order and explains its function. For a detailed description and specifics about writing procedures, refer to the [NetLogo Programming Guide](#).

Also, You can continue with this model if youd like, experimenting with different variables and algorithms to see what works the best (what makes the most turtles reach the peaks).

Alternatively, you can look at other models (including the many models in the Code Examples section of the Models Library) or even go ahead and build your own model. You dont even have to model anything. It can be pleasant just to watch patches and turtles forming patterns, or whatever. Hopefully you will have learned a few things, both in terms of syntax and general methodology for model- building. The entire code that was created above is shown below.

Appendix: Complete Code

The complete model is also available in NetLogos Models Library, in the Code Examples section. Its called Tutorial 3.

```

patches-own [ elevation ]      ;; elevation of the patch

globals [
  highest          ;; maximum patch elevation
  lowest           ;; minimum patch elevation
  turtles-moved?  ;; so we know when to stop the model
]

;; We also have two slider variables, 'number' and
;; 'smoothness'. 'number' determines the number of
;; turtles, and 'smoothness' determines how erratic
;; terrain becomes during diffusion of 'elevation'.

```

```

;; resets everything
to setup
  ca
  setup-patches
  setup-turtles
end

;; creates a random landscape of patch elevations
to setup-patches
  ask patches [set elevation (random 10000) ]
  repeat smoothness [diffuse elevation 1 ]
  ask patches
    [ set pcolor scale-color green elevation 1000 9000 ]

  set highest max values-from patches [elevation]
  set lowest min values-from patches [elevation]
  ask patches [
    if (elevation > (highest - 100))
      [set pcolor white]
    if (elevation <(lowest + 100))
      [set pcolor black]
  ]
end

;; initializes the turtles
to setup-turtles
  crt number
  ask turtles [
    if (color = green) [ set color red ]
    fd (random screen-edge-x)
  ]
end

;; RUN-TIME PROCEDURES
;; main program control
to go
  set turtles-moved? false
  move-to-local-max
  do-plots
  if (not turtles-moved?)
    [ stop ]
end

;; each turtle goes to the highest elevation in a radius of one
to move-to-local-max
  ask turtles [
    set heading uphill elevation
    if ( elevation-of patch-at dx dy > elevation )
      [
        fd 1
        set turtles-moved? true
      ]
  ]
end

to do-plots
  set-current-plot "Turtles at Peaks"
  plot count turtles with
    [ elevation >= (highest - 100) ]
end

```

Interface Guide

This section of the manual walks you through every element of the NetLogo interface in order and explains its function.

In NetLogo, you have the choice of or viewing models found in the Models Library, adding to existing models, or creating your own models. The NetLogo interface was designed to meet all these needs.

The interface can be divided into two main parts: NetLogo menus, and the main NetLogo window. The main window is divided into tabs.

- [Menus](#)
- [Main Window](#)
 - ◆ [Interface Tab](#)
 - ◇ Interface Toolbar
 - ◇ Working With Interface Elements
 - ◇ Graphics Window
 - ◇ Command Center
 - ◆ [Procedures Tab](#)
 - ◆ [Information Tab](#)
 - ◆ [Errors Tab](#)

Menus

On Macs, if you are running the NetLogo application, the menubar is located at the top of the screen. On other platforms, the menubar is found at the top of the NetLogo window.



The functions available from the menus in the menubar are listed in the following chart.

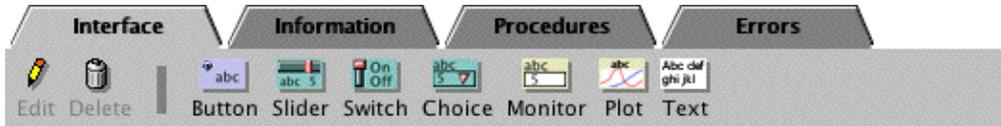
Chart: NetLogo Menus

File		
	New	Starts a new model.
	Open	Opens any NetLogo or StarLogoT model on your computer.
	Models Library	A collection of demonstration models.
	Save	Save the current model.
	Save As...	Save the current model using a different name.
	Save As Applet	Used to save a web page in HTML format that has your model embedded in it as a Java applet.
	Print	Sends the contents of the currently showing tab to your printer.
	Export World	Saves all variables and the current state of all turtles and patches to a file.
	Export Plot	Saves the data in a plot to a file.
	Export All Plots	Saves the data in all the plots to a file.
	Export Output	Save the contents of the output section of the command center to a file.

	Import World	Load a file that was saved by Export World.
	Quit	Exits NetLogo.
Edit		
	Undo	Undo that last action user performed.
	Cut	Cuts out or removes the selected text and temporarily saves it to the clipboard.
	Copy	Copies the selected text.
	Paste	Places the clipboard text where cursor is currently located.
	Delete	Deletes selected text.
	Find	Finds a word or sequence of characters within the Information, Procedures, or Errors tab.
	Find Again	Find the next occurrence of the word or sequence you last used Find with.
	Shift Left / Shift Right	Used in the Procedures and Errors tabs to change the indentation level of code.
	Comment / Uncomment	Used in the Procedures and Errors tabs to add or remove semicolons from code (semicolons are used in NetLogo code to indicate comments).
Tools		
	Halt	Stops all running code, including buttons and the command center. (Warning: since the code is interrupted in the middle of whatever it was doing, you may get unexpected results if you try to continue running the model without first pressing setup to start the model run over.)
	Globals Monitor	Displays the values of all global variables.
	Patch Monitor	Displays the values of all of the variables in a particular patch. You can also edit the values of the patch's variables and issue commands to the patch. (You can also open a patch monitor via the Graphics Window; see the Graphics Window section below.)
	Turtle Monitor	Displays the values of all of the variables in a particular turtle. You can also edit the values of the turtle's variables and issue commands to the turtle. (You can also open a turtle monitor via the Graphics Window; see the Graphics Window section below.)
	Shapes Editor	Draw turtle shapes. See the Shapes Editor Guide for more information.
	BehaviorSpace	Runs the model over and over with different settings. See the BehaviorSpace Guide for more information.
Zoom		
	Larger	Increase the overall screen size of the model. Useful on large monitors or when using a projector in front of a group.
	Normal Size	Reset the screen size of the model to the normal size.
	Smaller	Decrease the overall screen size of the model.
Tabs		This menu offers keyboard shortcuts for each of the tabs. (On Macs, it's Command 1 through Command 4. On Window, it's Control 1 through Control 4.)
Help		
	About NetLogo	Information on the current NetLogo version the user is running. (On Macs, this menu item is on the Apple menu instead.)
	User Manual	Opens this manual in a web browser.

Main Window

At the top of NetLogo's main window are four tabs labeled Interface, Procedure, Information, and Errors. Only one tab at a time can be visible, but you can switch between them by clicking on the tabs at the top of the window.



Right below the row of tabs is a toolbar containing a row of buttons. The buttons available vary from tab to tab.

Interface Tab

The Interface Tab is where you watch your model run. It also has tools you can use to inspect and alter what's going on inside the model.

When you first open NetLogo, the Interface tab is empty except for the Graphics Window, where the turtles and patches appear, and the Command Center, which allows you to issue NetLogo commands.

Interface Toolbar

The toolbar contains buttons that let you edit, delete, and create items in the Interface tab (such as buttons and sliders).



The buttons in the toolbar are described below.

Working With Interface Elements

Selecting: To select an interface element, drag a rectangle around it with your mouse. A gray border will appear around the element to indicate that it is selected.

Selecting Multiple Items: You can select multiple interface elements at the same time by including them in the rectangle you drag. If multiple elements are selected, one of them is the key item, which means that if you use the Edit or Delete buttons on the Interface Toolbar, only the key item is affected. The key item is indicated by a darker gray border than the other items.

Unselecting: To unselect all interface elements, click the mouse on the white background of the Interface tab. To unselect an individual element, control-click (Macintosh) or right-click (other OS) the element and choose Unselect from the popup menu.

Editing: To change the characteristics of an interface element, select the element, then press the Edit button on the Interface Toolbar. You may also double click the element once it is selected. A third way to edit an element is to control-click (Macintosh) or right-click (other OS) it and choose

Edit from the popup menu. If you use this last method, it is not necessary to select the element first.

Moving: Select the interface element, then drag it with your mouse to its new location. If you hold down the shift key while dragging, the element will move only straight up and down or straight left and right.

Resizing: Select the interface element, then drag the black handles in the selection border.

Deleting: Select the element or elements you want to delete, then press the Delete button on the Interface Toolbar. You may also delete an element by control-clicking (Macintosh) or right-clicking (other OS) it and choosing Delete from the popup menu. If you use this latter method, it is not necessary to select the element first.

To learn more about the different kinds of interface elements, refer to the chart below.

Chart: Interface Toolbar

Icon & Name	Description
 <p>Button</p>	Buttons can be either <i>once-only</i> buttons or <i>forever</i> buttons. When you click on a once-button, it executes its instructions once. The forever-button executes the instructions over and over, until you click on the button again to stop the action.
 <p>Slider</p>	Sliders are global variables, which are accessible by all agents. They are used in models as a quick way to change a variable without having to recode the procedure every time. Instead, the user moves the slider to a value and observe what happens in the model.
 <p>Switch</p>	Switches are a visual representation for a true/false variable. The user is asked to set the variable to either on (true) or off (false) by flipping the switch.
 <p>Choice</p>	Choices let the user pick a value for a global variable from a list of choices, presented in a drop down menu.
 <p>Monitor</p>	Monitors display the value of any expression. The expression could be a variable, a complex expression, or a call to a reporter. Monitors automatically update several times per second.
 <p>Plot</p>	Plots are real-time graphs of data the model is generating.
 <p>Text</p>	A Text Box lets you create text labels in the Interface Tab.

Graphics Window

The Graphics Window initially appears as a large black square on the Interface Tab. This is the graphical world of NetLogos turtles and patches.

Some NetLogo models let you interact with the turtles and patches with your mouse by clicking and dragging in the Graphics Window.

The Graphics Window provides an easy way to open a turtle monitor or patch monitor. Just control-click (Macintosh) or right-click (other OS) on the turtle or patch you want to inspect, and

choose inspect turtle ... or inspect patch ... from the popup menu. (Turtle and patch monitors can also be opened from the Tools menu or by using the `inspect` command.)

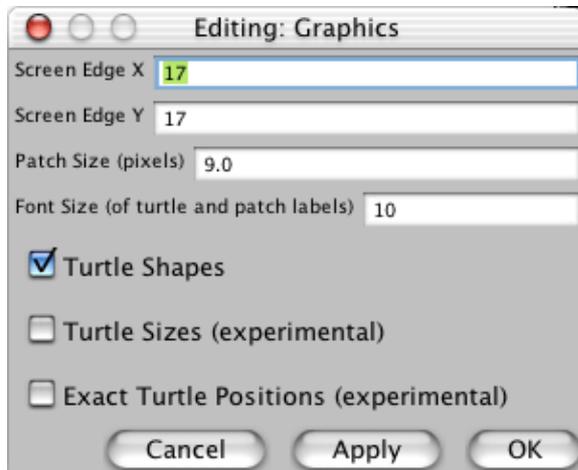
There are a number of settings associated with the Graphics Window. There are two ways of changing the settings: by using the control strip along the top edge of the Graphics Window, or by editing the Graphics Window, as described in the Working With Interface Elements section above. (Or, pressing the More... button in the control strip is an alternate and quicker way.)



The controls in the control strip work as follows:

- The three sets of black arrows let you change the size of the world.
- The slider lets you control how fast the model runs this is valuable since some models run so fast that its hard to see whats going on.
- The button with the arrowhead lets you turn turtle shapes on and off. If shapes are off, turtles appear as colored squares, instead of having special shapes. The squares are less work for the computer to draw, so turning shapes off makes models run faster.
- The on-off switch lets you temporarily freeze the display. The model keeps running, but the contents of the graphics window don't change until you unfreeze it by flipping the switch again. Most models run much faster when the graphics window is frozen.

Here are the settings for the Graphics Window (accessible by editing the Graphics Window, or by pressing the More... button in the control strip):



To change the size of the Graphics Window adjust the Patch Size setting, which is measured in pixels. This does not change the number of patches, only how large the patches appear on the screen. To change the number of patches, alter the Screen Edge X and Screen Edge Y settings. (Note that changing the numbers of patches requires rebuilding the NetLogo world; you will lose all turtles and the values of all variables.)

In most NetLogo models, every turtle is always the same size: the same size as all the patches. The Turtle Sizes checkbox lets you turn on an optional feature where turtles can be drawn in all different sizes. The size of each turtle is controlled by its `size` turtle variable.

In most NetLogo models, every turtle is always drawn as if it were standing on the center of its patch. The Exact Turtle Positions checkbox lets you turn on an optional feature where turtles can be drawn in their exact positions within a patch.

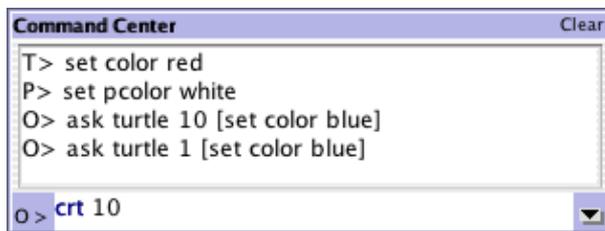
Note: the Turtle Sizes and Exact Turtle Positions features may make your model run a lot slower and/or cause display anomalies. We plan to improve these features in a future version of NetLogo.

Command Center

The Command Center allows you to issue commands directly, without adding them to the models procedures. (Commands are instructions you give to turtles, patches, and the observer.) This is useful for inspecting and manipulating agents on the fly.

([Tutorial #2: Commands](#) is an introduction to using commands in the Command Center.)

Lets take a closer look at the design of the Command Center.



You will notice there is a large display box, an agent popup menu (O>), a clear button, and the history popup menu (with the little black triangle). The top large display box temporarily stores all of the commands that are entered into the Command Center. This area is strictly for reference; commands cannot be accessed or changed from this box. To clear this box, click clear in the top right corner.

The smaller text box, below the large box, is where commands are entered. On the left of this box is the agent popup menu, and on the right is the history popup menu.

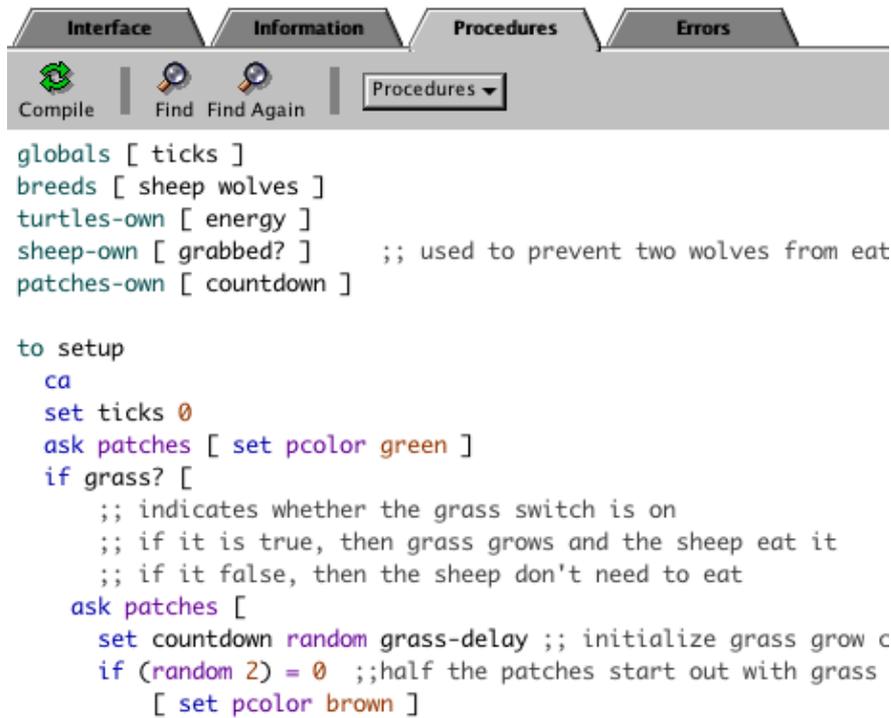
The agent popup menu allows you to select either observer, turtles, or patches. This is an easy way to assign an agent to a command and it is the same as writing `ask turtles [...]`. Note: a quicker way to change between observer, turtles, and patches is to use the tab key on your keyboard.

The history popup menu lists all of the commands entered that can be accessed and used again. The up and down arrow keys on your keyboard will retrieve that last command that was written.

Note that pressing the clear function clears only the large display box and not the history. To clear the history section, choose clear history, found at the top of its popup menu.

Procedures Tab

This tab is the workspace where the code for the model is stored. Commands you only want to use immediately go in the Command Center; commands you want to save and use later, over and over again, are found in the Procedures tab.



To determine if the code has any errors, press the Compile button. If there are any syntax errors, the Errors Tab will come to the front of the screen and turn red. The code that contains the error will be highlighted and a comment will appear in the top box. Switching tabs also causes the code to be compiled and any errors will be shown, so if you switch tabs, pressing the Compile button first isn't necessary.

To find a fragment of code in the procedures, click on the magnifying glass on the Procedures Toolbar. Then enter the text you are looking for, and hit the Find button. The Find Again button finds the next location of the word or fragment of code throughout the procedure.

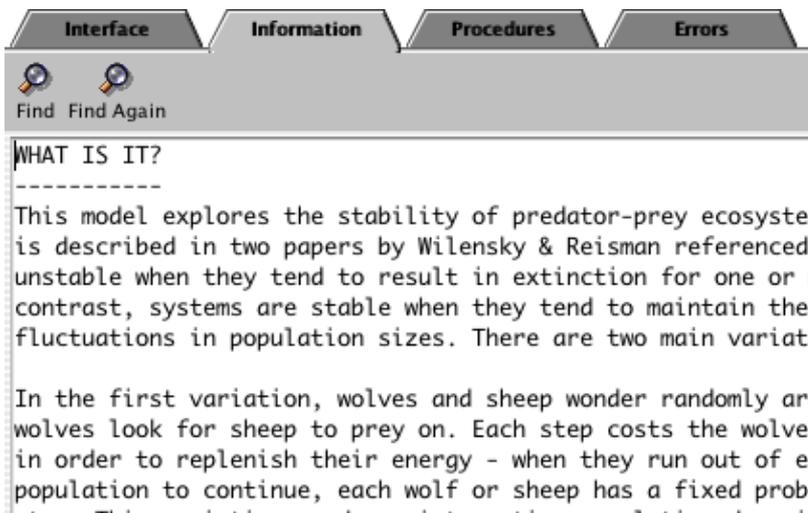
To find a particular procedure definition in your code, use the Procedures popup menu in the Procedures Toolbar. The menu lists all procedures in alphabetical order.

The Shift Left, Shift Right, Comment, and Uncomment items on the Edit menu are used in the procedures tab to change the indentation level of your code or add and remove semicolons, which mark comments, from sections of code.

For more information about writing procedures, read [Tutorial #3: Procedures](#) and the [Programming Guide](#).

Information Tab

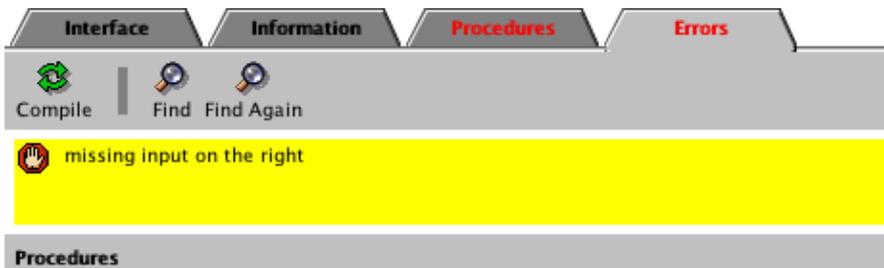
The Information tab provides an introduction to the model and an explanation of how to use it, things to explore, possible extensions, and NetLogo features. It is very helpful when you're first exploring a model.



We recommend reading the Information Tab before starting the model. The Information Tab explains what principle is being modeled and how the model was created.

Errors Tab

The Errors Tab provides a place for errors to be highlighted and brought to your attention. If there are any errors in your code, the Errors Tab will turn red and come to the front with an explanation of what caused the error.



```
set-default-shape wolves "wolf-shape"
create-custom-wolves init-wolves ;; create the wolves, then ini
[
  set color
  set energy random (2 * wolf-metabolism)
  setxy random screen-size-x random screen-size-y
]
```

Switching tabs, or pressing the Compile button in the toolbar, causes your code to be checked for errors.

Programming Guide

The following material explains some important features of programming in NetLogo.

(Note: If you are already familiar with StarLogo or StarLogoT, then the material in the first four sections may already be familiar to you.)

The Code Example models mentioned throughout can be found in the Code Examples section of the Models Library.

- [Agents](#)
- [Procedures](#)
- [Variables](#)
- [Colors](#)
- [Ask](#)
- [Agentsets](#)
- [Breeds](#)
- [Synchronization](#)
- [Procedures \(advanced\)](#)
- [Lists](#)
- [Strings](#)
- [Turtle Shapes](#)

Agents

The NetLogo world is made up of agents. Agents are beings that can follow instructions. Each agent can carry out its own activity, all simultaneously.

In NetLogo, there are three types of agents: turtles, patches, and the observer. Turtles are agents that move around in the world. The world is two dimensional and is divided up into a grid of patches. Each patch is a square piece of ground over which turtles can move. The observer doesn't have a location you can imagine it as looking out over the world of turtles and patches.

When NetLogo starts up, there are no turtles yet. The observer can make new turtles. Patches can make new turtles too. (Patches can't move, but otherwise they're just as alive as turtles and the observer are.)

Patches have coordinates. The patch in the center of the world has coordinates (0, 0). We call the patch's coordinates `pxcor` and `pycor`. Just like in the standard mathematical coordinate plane, `pxcor` increases as you move to the right and `pycor` increases as you move up.

The total number of patches is determined by the settings `screen-edge-x` and `screen-edge-y`. When NetLogo starts up, both `screen-edge-x` and `screen-edge-y` are 17. This means that `pxcor` and `pycor` both range from -17 to 17, so there are 35 times 35, or 1225 patches total. (You can change the number of patches by editing NetLogo's Graphics window.)

Turtles have coordinates too: `xcor` and `ycor`. A patch's coordinates are always integers, but a turtle's coordinates can have decimals. For speed, NetLogo always draws a turtle on-screen as if it were standing in the center of its patch, but in fact, the turtle can be positioned at any point within

the patch.

The world of patches isn't bounded, but wraps so when a turtle moves past the edge of the world, it disappears and reappears on the opposite edge. Every patch has the same number of neighbor patches if you're a patch on the edge of the world, some of your neighbors are on the opposite edge.

Procedures

In NetLogo, commands and reporters tell agents what to do. **Commands** are actions for the agents to carry out. **Reporters** carry out some operation and report a result either to a command or another reporter.

Commands and reporters built into NetLogo are called **primitives**. [The Primitives Dictionary](#) has a complete list of built-in commands and reporters.

Commands and reporters you define yourself are called **procedures**. Each procedure has a name, preceded by the keyword `to`. The keyword `end` marks the end of the commands in the procedure. Once you define a procedure, you can use it elsewhere in your program.

Many commands and reporters take **inputs** values that the command or reporter uses in carrying out its actions.

Examples: Here are two command procedures:

```
to setup
  ca          ;; clear the screen
  crt 10      ;; make 10 new turtles
end

to go
  ask turtles
  [ fd 1      ;; all turtles move forward one step
    rt random 10  ;; ...and turn a random amount
    lt random 10 ]
end
```

Note the use of semicolons to add comments to the program. Comments make your program easier to read and understand.

In this program,

- `setup` and `go` are user-defined commands.
- `ca` (clear all), `crt` (create turtles), `ask`, `lt` (left turn), and `rt` (right turn) are all primitive commands.
- `random` and `turtles` are primitive reporters. `random` takes a single number as an input and reports a random number that is less than the input (in this case, between 0 and 9). `turtles` reports the agentset consisting of all the turtles. (We'll explain about agentsets later.)

`setup` and `go` can be called by other procedures or by buttons. Many NetLogo models have a once-button that calls a procedure called `setup`, and a forever-button that calls a procedure called

go.

In NetLogo, you must specify which agents—turtles, patches, or the observer—are to run each command. (If you don't specify, the code is run by the observer.) In the code above, the observer uses `ask` to make the set of all turtles run the commands between the square brackets.

`ca` and `cr` can only be run by the observer. `fd`, on the other hand, can only be run by turtles. Some other commands and reporters, such as `set`, can be run by different agent types.

Variables

Variables are places to store values (such as numbers). A variable can be a global variable, a turtle variable, or a patch variable.

If a variable is a global variable, there is only one value for the variable, and every agent can access it. But each turtle has its own value for every turtle variable, and each patch has its own value for every patch variable.

Some variables are built into NetLogo. For example, all turtles have a `color` variable, and all patches have a `pcolor` variable. (The patch variable begins with `p` so you won't get it confused with the turtle variable.) If you set the variable, the turtle or patch changes color. (See next section for details.)

Other built-in turtle variables including `xcor`, `ycor`, and `heading`. Other built-in patch variables include `pxcor` and `pycor`. (There is a complete list [here](#).)

You can also define your own variables. You can make a global variable by adding a switch or a slider to your model, or by using the [globals](#) keyword at the beginning of your code, like this:

```
globals [ clock ]
```

You can also define new turtle and patch variables using the [turtles-own](#) and [patches-own](#) keywords, like this:

```
turtles-own [ energy speed ]
patches-own [ friction ]
```

These variables can then be used freely in your model. Use the [set](#) command to set them. (If you don't set them, they'll start out storing a value of zero.)

Global variables can be read and set at any time by any agent. As well, a turtle can read and set patch variables of the patch it is standing on. For example, this code:

```
ask turtles [ set pcolor red ]
```

causes every turtle to make the patch it is standing on red.

In other situations where you want an agent to read or set a different agent's variable, you put `-of` after the variable name and then specify which agent you mean. Examples:

```
set color-of turtle 5 red
;; turtle with ID number 5 turns red
```

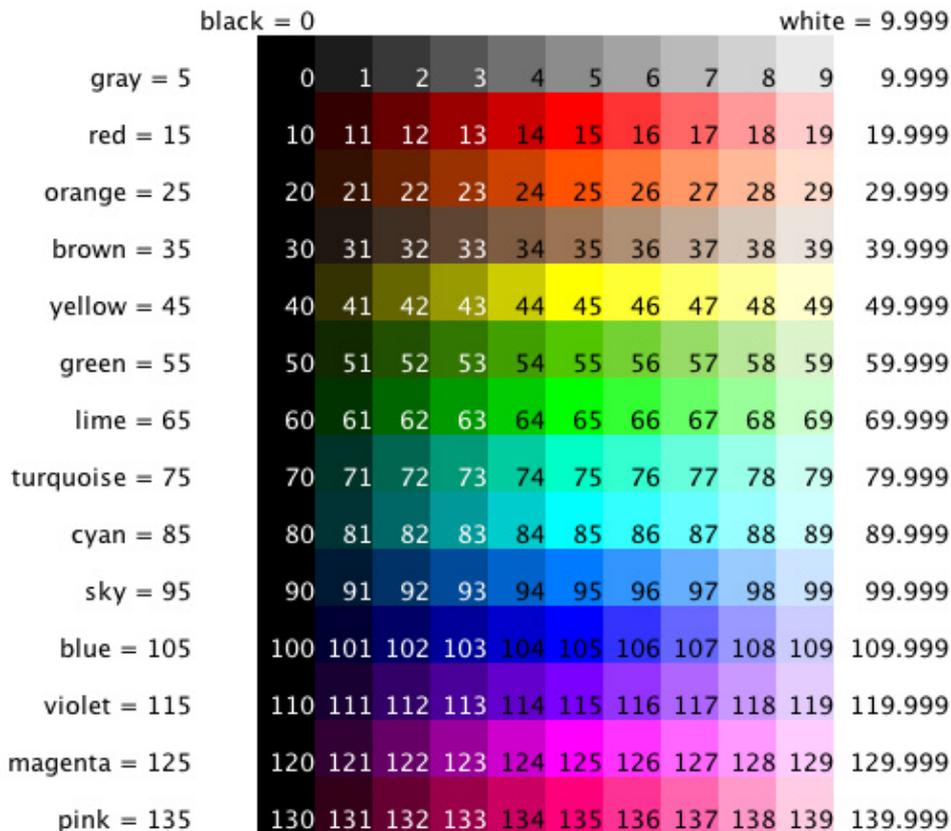
```

set pcolor-of patch 2 3 green
;; patch with pxcor of 2 and pycor of 3 turns green
ask turtles [ set pcolor-of patch-at 1 0 blue ]
;; every turtle turns the patch to its east blue
ask patches with [any turtles-here]
  [ set color-of random-one-of turtles-here yellow ]
;; on every patch, a random turtle turns yellow

```

Colors

NetLogo represents colors as numbers in the range 0 to 140, with the exception of 140 itself. Below is a chart showing the range of colors you can use in NetLogo.



The chart shows that:

- Some of the colors have names. (You can use these names in your code.)
- Every named color except black and white has a number ending in 5.
- On either side of each named color are darker and lighter shades of the color.
- 0, 10, 20, and so on are all black. 9.999, 19.999, 29.999 and so on are all white.

Code Example: The color chart was made in NetLogo with the Color Chart Example model.

Note: If you use a number outside the 0 to 140 range, NetLogo will repeatedly add or subtract 140 from the number until it is in the 0 to 140 range. For example, 25 is orange, so 165, 305, 445, and

so on are orange too, and so are -115 , -255 , -395 , etc. This calculation is done automatically whenever you set the turtle variable `color` or the patch variable `pcolor`. Should you need to perform this calculation in some other context, use the [wrap-color](#) primitive.

If you want a color that's not on the chart, more can be found between the integers. For example, 26.5 is a shade of orange halfway between 26 and 27 . This doesn't mean you can make any color in NetLogo; the NetLogo color space is only a subset of all possible colors. A fixed set of discrete hues is available, and you can either decrease the brightness (darken) or decrease the saturation (lighten) of those hues to get your desired color, but you may not decrease both the brightness and saturation.

There are a few primitives that are helpful for working with color shades. The [scale-color](#) primitive is useful for converting numeric data into colors. And [shade-of?](#) will tell you if two colors are shades of the same basic hue. For example, `shade-of? orange 27` is true, because 27 is a lighter shade of orange.

Code Example: Scale-color Example shows you how to use the scale-color reporter.

For many models, the NetLogo color system is a convenient way of expressing colors. But sometimes you'd like to be able to specify colors the conventional way, by specifying HSB (hue/saturation/brightness) or RGB (red/green/blue) values. The [hsb](#) and [rgb](#) primitives let you do this. [extract-hsb](#) and [extract-rgb](#) let you convert colors in the other direction.

Since the NetLogo color space doesn't include all hues, `hsb` and `rgb` can't always give you the exact color you ask for, but they try to come as close as possible.

Code Example: You can use the HSB and RGB Example model to experiment with the HSB and RGB color systems.

Ask

NetLogo uses the [ask](#) command to specify commands that are to be run by turtles or patches. All code to be run by turtles **must** be located in a turtle context. You can establish a turtle context in any of three ways:

- In a button, by choosing Turtles from the popup menu. Any code you put in the button will be run by all turtles.
- In the Command Center, by choosing Turtles from the popup menu. Any commands you enter will be run by all the turtles.
- By using `ask turtles`.

The same goes for patches and the observer, except that code to be run by the observer must not be inside any `ask`.

Here's an example of the use of `ask` syntax in a NetLogo procedure:

```
to setup
  ca
```

```

crt 100          ;; create 100 turtles
ask turtles
  [ set color red      ;; turn them red
    rt random 360      ;; give them random headings
    fd 50 ]          ;; spread them around
ask patches
  [ if (pxcor > 0)      ;; patches on the right side
    [ set pcolor green ] ] ;; of the screen turn green
end

```

The models in the Models Library are full of other examples. A good place to start looking is in the Code Examples section.

Usually, the observer uses `ask` to ask all turtles or all patches to run commands. You can also use `ask` to have make an individual turtle or patch run commands. The reporters [turtle](#), [patch](#), and [patch-at](#) are useful for this technique. For example:

```

to setup
  ca
  crt 3          ;; make 3 turtles
  ask turtle 0   ;; tell the first one...
  [ fd 1 ]      ;; ...to go forward
  ask turtle 1   ;; tell the second one...
  [ set color green ] ;; ...to become green
  ask turtle 2   ;; tell the third one...
  [ rt 90 ]     ;; ...to turn right
  ask patch 2 -2 ;; ask the patch at coords (2,-2)
  [ set pcolor blue ] ;; ...to become blue
  ask turtle 0   ;; ask the first turtle
  [ ask patch-at 1 0 ;; ...to ask patch to the east
    [ set pcolor red ] ;; ...to become red
  ]
end

```

Every turtle created has an ID number. The first turtle created has ID 0, the second turtle ID 1, and so forth. The `turtle` primitive reporter takes an ID number as an input, and reports the turtle with that ID number. The `patch` primitive reporter takes values for `pxcor` and `pycor` and reports the patch with those coordinates. And the `patch-at` primitive reporter takes *offsets*: distances, in the x and y directions, *from* the first agent. In the example above, the turtle with ID number 0 is asked to get the patch east (and no patches north) of itself.

You can also select a subset of turtles, or a subset of patches, and ask them to do something. This involves a concept called agentsets. The next section explains this concept in detail.

Agentsets

An agentset is exactly what its name implies, a set of agents. An agentset can contain either turtles or patches, but not both at once.

You've seen the `turtles` primitive, which reports the agentset of all turtles, and the `patches` primitive, which reports the agentset of all patches.

But what's powerful about the agentset concept is that you can construct agentsets that contain only *some* turtles or *some* patches. For example, all the red turtles, or the patches with `pxcor` evenly divisible by five, or the turtles in the first quadrant that are on a green patch. These agentsets can then be used by `ask` or by various reporters that take agentsets as inputs.

One way is to use [turtles-here](#) or [turtles-at](#) to make an agentset containing only the turtles on my patch, or only the turtles on some other particular patch.

Here are some more examples of how to make agentsets:

```
;; all red turtles:
turtles with [color = red]
;; all red turtles on my patch
turtles-here with [color = red]
;; patches on right side of screen
patches with [pxcor > 0]
;; all turtles less than 3 patches away
turtles in-radius 3
;; the four patches to the east, north, west, and south
patches at-points [[1 0] [0 1] [-1 0] [0 -1]]
;; shorthand for those four patches
neighbors4
;; turtles in the first quadrant that are on a green patch
turtles with [(xcor > 0) and (ycor > 0)
              and (pcolor = green)]
```

Once you have created an agentset, here are some simple things you can do:

- Use [ask](#) to make the agents in the agentset do something
- Use [any](#) to see if the agentset is empty
- Use [count](#) to find out exactly how many agents are in the set

And here are some more complex things you can do:

- Pick a random agent from the set using [random-one-of](#). For example, we can make a randomly chosen turtle turn green:

```
set color-of random-one-of turtles green
```

Or tell a randomly chosen patch to [sprout](#) a new turtle:

```
ask random-one-of patches [ sprout 1 [ ] ]
```

- Use the [max-one-of](#) or [min-one-of](#) reporters to find out which agent is the most or least along some scale. For example, to remove the richest turtle, you could say

```
ask max-one-of turtles [sum assets] [ die ]
```

- Make a histogram of the agentset using the [histogram](#) command.
- Use [values-from](#) to make a list of values, one for each agent in the agentset. Then use one of NetLogo's list primitives to do something with the list. (See the Lists section [below](#).) For example, to find out how rich the richest turtle is, you could say

```
show max values-from turtles [sum assets]
```

This only scratches the surface see the Models Library for many more examples, and consult the Primitives Guide and Primitives Dictionary for more information about all of the agentset primitives.

More examples of using agentsets are provided in the individual entries for these primitives in the NetLogo Dictionary. In developing familiarity with programming in NetLogo, it is important to begin

to think of compound commands in terms of how each element passes information to the next one. Agentsets are an important part of this conceptual scheme and provide the NetLogo developer with a lot of power and flexibility, as well as being more similar to natural language.

Code Example: Ask Agentset Example

Breeds

NetLogo allows you to define different breeds of turtles. Once you have defined breeds, you can go on and make the different breeds behave differently. For example, you could have breeds called `sheep` and `wolves`, and have the wolves try to eat the sheep.

You define breeds using the [breeds](#) keyword, at the top of your model, before any procedures:

```
breeds [wolves sheep]
```

When you define a breed such as `sheep`, an agentset for that breed is automatically created, so that all of the agentset capabilities described above are immediately available with the `sheep` agentset.

The following new primitives are also automatically available once you define a breed: [create-sheep](#), [create-custom-sheep](#) (`cct-sheep` for short), [sheep-here](#), and [sheep-at](#).

Also, you can use [sheep-own](#) to define new turtle variables that only turtles of the given breed have.

A turtles breed agentset is stored in the `breed` turtle variable. So you can test a turtles breed, like this:

```
if breed = wolves [ ... ]
```

Note also that turtles can change breeds. A wolf doesn't have to remain a wolf its whole life. Let's change a random wolf into a sheep:

```
ask random-one-of wolves [ set breed sheep ]
```

The [set-default-shape](#) primitive is useful for associating certain turtle shapes with certain breeds. See the section on shapes [below](#).

Here is a quick example of using breeds:

```
breeds [mice frogs]
mice-own [cheese]
to setup
  ca
  create-custom-mice 50
  [ set color white
    set cheese random 10 ]
  create-custom-frogs 50
  [ set color green ]
end
```

Code Example: Breeds and Shapes Example

Synchronization

In both StarLogoT and NetLogo, commands are executed asynchronously; each turtle or patch does its list of commands as fast as it can. In StarLogoT, one could make the turtles line up by putting in a comma (,). At that point, the turtles would wait until all were finished before any went on.

The equivalent in NetLogo is to come to the end of an ask block. If you write it this way, the two steps are not synced:

```
ask turtles
  [ fd random 10
    do-calculation ]
```

But if you write it this way, they are:

```
ask turtles [ fd random 10 ]
ask turtles [ do-calculation ]
```

This latter form is equivalent to this use of the comma in StarLogoT:

```
fd random 10 ,
do-calculation ,
```

Procedures (advanced)

Here are some more advanced features you can take advantage of when defining your own procedures.

Procedures with inputs

Your own procedures can take inputs, just like primitives do. To create a procedure that accepts inputs, include a list of input names in square brackets after the procedure name. For example:

```
to draw-polygon [num-sides size]
  pd
  repeat num-sides
    [ fd size
      rt (360 / num-sides) ]
end
```

Elsewhere in the program, you could ask turtles to each draw an octagon with a side length equal to its ID-number:

```
ask turtles [ draw-polygon 8 who ]
```

Reporter procedures

Just like you can define your own commands, you can define your own reporters. You must do two special things. First, use [to-report](#) instead of `to` to begin your procedure. Then, in the body of the procedure, use [report](#) to report the value you want to report.

```
to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report 0 - number ]
end
```

Procedures with local variables

A local variable is defined and used only in the context of a particular procedure. To add a local variable to your procedure, use the [locals](#) keyword. It must come at the beginning of your procedure. For example:

```
to swap-colors [turtle1 turtle2]
  locals [temp]
  set temp color-of turtle1
  set (color-of turtle1) (color-of turtle2)
  set (color-of turtle2) temp
end
```

Lists

In the simplest models, each variable holds only one piece of information, usually a number or a string. The list feature lets you store multiple pieces of information in a single variable by collecting those pieces of information in a list. Each value in the list can be any type of value: a number, or a string, an agent or agentset, or even another list.

Lists allow for the convenient packaging of information in NetLogo. If your agents carry out a repetitive calculation on multiple variables, it might be easier to have a list variable, instead of multiple number variables. Several primitives simplify the process of performing the same computation on each value in a list.

The [Primitives Dictionary](#) has a section that lists of all the list-related primitives.

Constant Lists

You can make a list by simply putting the values you want in the list between brackets, like this: `set mylist [2 4 6 8]`. Note that the individual values are separated by spaces. You can make lists that contains numbers and strings this way, as well as lists within lists, for example `[[2 4] [3 5]]`.

The empty list is written by putting nothing between the brackets, like this: `[]`.

Building Lists on the Fly

If you want to make a list in which the values are determined by reporters, as opposed to being a series of constants, use the [list](#) reporter. The `list` reporter accepts two other reporters, runs them, and reports the results as a list.

If I wanted a list to contain two random values, I might use the following code:

```
set random-list list (random 10) (random 20)
```

This will set `random-list` to a new list of two random numbers each time it runs.

To make longer lists, use the `list` reporter with the [sentence](#) reporter, which concatenates two lists (combines their contents into a single, larger list).

The [values-from](#) primitive lets you construct a list from an agentset. It reports a list containing the each agents value for the given reporter. (The reporter could be a simple variable name, or a more complex expression even a call to a procedure defined using `to-report`.) A common idiom is

```
max values-from turtles [...]
sum values-from turtles [...]
```

and so on.

Changing List Items

Technically, only one command changes a list `set`. This is used in conjunction with reporters. For example, to change the third item of a list to 10, you could use the following code:

```
set mylist [2 7 5 Bob [3 0 -2]]
; mylist is now [2 7 5 Bob [3 0 -2]]
set mylist replace-item 2 mylist 10
; mylist is now [2 7 10 Bob [3 0 -2]]
```

The [replace-item](#) reporter takes three inputs. The first input specifies which item in the list is to be changed. 0 means the first item, 1 means the second item, and so forth.

To add an item, say 42, to the end of a list, use the [lput](#) reporter. ([fput](#) adds an item to the beginning of a list.)

```
set mylist lput 42 mylist
; mylist is now [2 7 10 Bob [3 0 -2] 42]
```

But what if you changed your mind? The [but-last](#) (`bl` for short) reporter reports all the list items but the last.

```
set mylist but-last mylist
; mylist is now [2 7 10 Bob [3 0 -2]]
```

Suppose you want to get rid of item 0, the 2 at the beginning of the list.

```
set mylist but-first mylist
; mylist is now [7 10 Bob [3 0 -2]]
```

Suppose you wanted to change the third item that's nested inside item 3 from `-2` to `9`? The key is to realize that the name that can be used to call the nested list `[3 0 -2]` is `item 3 mylist`. Then the `replace-item` reporter can be nested to change the list-within-a-list. The parentheses are added for clarity.

```
set mylist (replace-item 3 mylist
            (replace-item 2 (item 3 mylist) 9))
; mylist is now [7 10 Bob [3 0 9]]
```

Iterating Over Lists

If you want to do some operation on each item in a list in turn, the [foreach](#) command and the [map](#)

reporter may be helpful.

`foreach` is used to run a command or commands on each item in a list. It takes an input list and a block of commands, like this:

```
foreach [2 4 6]
  [ crt ?
    show "created " + ? + " turtles" ]
=> created 2 turtles
=> created 4 turtles
=> created 6 turtles
```

In the block, the variable `?` holds the current value from the input list.

Here are some more examples of `foreach`:

```
foreach [1 2 3] [ ask turtles [ fd ? ] ]
;; turtles move forward 6 patches
foreach [true false true true] [ ask turtles [ if ? [ fd 1 ] ] ]
;; turtles move forward 3 patches
```

`map` is similar to `foreach`, but it is a reporter. It takes an input list and another reporter. Note that unlike `foreach`, the reporter comes first, like this:

```
show map [round ?] [1.2 2.2 2.7]
;; prints [1 2 3]
```

`map` reports a list containing the results of applying the reporter to each item in the input list. Again, use `?` to refer to the current item in the list.

Here is another example of `map`:

```
show map [? < 0] [1 -1 3 4 -2 -10]
;; prints [false true false false true true]
```

`foreach` and `map` won't necessarily be useful in every situation in which you want to operate on an entire list. In some situations, you may need to use some other technique such as a loop using `repeat` or `while`, or a recursive procedure.

The `sort-by` primitive uses a similar syntax to `map` and `foreach`, except that since the reporter needs to compare two objects, the two special variables `?1` and `?2` are used in place of `?`.

Here is an example of `sort-by`:

```
show sort-by [?1 < ?2] [4 1 3 2]
;; prints [1 2 3 4]
```

Varying Numbers of Inputs

Some commands and reporters involving lists and strings may take a varying number of inputs. In these cases, in order to pass them a number of inputs other than their default, the primitive and its inputs must be surrounded by parentheses. Here are some examples:

```
show list 1 2
=> [1 2]
```

```
show (list 1 2 3 4)
=> [1 2 3 4]
show (list)
=> []
```

Note that each of these special commands has a default number of inputs for which no parentheses are required. The primitives which have this capability are [list](#), [word](#), [sentence](#), [map](#), and [foreach](#).

Strings

To input a constant string in NetLogo, surround it with double quotes.

The empty string is written by putting nothing between the quotes, like this: " ".

Most of the list primitives work on strings as well:

```
butfirst "string" => "tring"
butlast "string" => "strin"
empty? "" => true
empty? "string" => false
first "string" => "s"
item 2 "string" => "r"
last "string" => "g"
length "string" => 6
member? "s" "string" => true
member? "rin" "string" => true
member? "ron" "string" => false
position "s" "string" => 0
position "rin" "string" => 2
position "ron" "string" => false
remove "r" "string" => "sting"
remove "s" "strings" => "tring"
replace-item 3 "string" "o" => "strong"
reverse "string" => "gnirts"
```

A few primitives are specific to strings, such as [is-string?](#), [substring](#), and [word](#):

```
is-string? "string" => true
is-string? 37 => false
substring "string" 2 5 => "rin"
word "tur" "tle" => "turtle"
```

Strings can be compared using the =, !=, <, >, <=, and >= operators.

To concatenate strings, that is, combine them into a single string, you may also use the + (plus) operator, like this:

```
"tur" + "tle" => "turtle"
```

If you need to embed a special character in a string, use the following escape sequences:

- \n = newline (carriage return)
- \t = tab
- \" = double quote
- \\ = backslash

Turtle shapes

In StarLogoT, turtle shapes were bitmaps. They all had a single fixed size and could only rotate in 45 degree increments.

In NetLogo, turtle shapes are vector shapes. They are built up from basic geometric shapes; squares, circles, and lines, rather than a grid of pixels. Vector shapes are fully scalable and rotatable.

A turtles shape is stored in its `shape` variable and can be set using the `set` command.

New turtles have a shape of default. The [set-default-shape](#) primitive is useful for changing the default turtle shape to a different shape, or having a different default turtle shape for each breed of turtle.

Use the Shapes Editor to create your own turtle shapes. For more information, see the Shapes Editor [section](#) of this manual.

Code Examples: Breeds and Shapes Example, Shape Animation Example

HubNet Guide

HubNet is a technology that lets you use NetLogo to run *participatory simulations* in the classroom. In a participatory simulation, a whole class takes part in enacting the behavior of a system as each student controls a part of the system by using an individual device, such as a TI-83+ calculator or a networked computer.

For example, in the Gridlock simulation, each student controls a traffic light in a simulated city. The class as a whole tries to make traffic flow efficiently through the city. As the simulation runs, data is collected which can afterwards be analyzed on a calculator or computer.

For more information on participatory simulations and their learning potential, please visit the [Participatory Simulations Project web site](#).

- [About HubNet](#)
 - ◆ [HubNet Types](#)
 - ◆ [What do I need to get started?](#)
 - ◇ [Calculator HubNet](#)
 - ◇ [Computer HubNet](#)
 - ◆ [First-time NetLogo user?](#)
 - ◆ [Teacher workshops](#)
- [Getting Started With HubNet](#)
 - ◆ [How to use NetLogo](#)
 - ◆ [About the activities](#)
 - ◆ [Running an activity](#)
 - ◇ [Calculator HubNet](#)
 - ◇ [Computer HubNet](#)
- [HubNet Authoring Guide](#)
- [Computer HubNet Troubleshooting Tips](#)
- [Known Computer HubNet Issues](#)

About HubNet

HubNet Types

Currently, there are two types of HubNet available. One, called Calculator HubNet, was created in conjunction with Texas Instruments and makes use of the TI-Navigator system. This type of HubNet uses TI-83+ graphing calculators as the clients, the devices used to control a portion of a NetLogo model.

The second type of HubNet is Computer HubNet. This is adapted from the calculator version and uses laptop or desktop networked computers as the clients.

In the future, we hope to add more types of HubNet that support other types of clients such as PDAs (Personal Digital Assistants).

What do I need to get started?

Calculator HubNet

- **A computer with an attached projector.** This computer will run NetLogo and project the simulation for class viewing.
- **A classroom set of TI-83+ graphing calculators.**
- **TI-Navigator wireless calculator network.**

NOTE: TI-Navigator recently became commercially available. To learn more about the TI-Navigator system, visit [Texas Instruments site](#). However, HubNet does not work with this version of TI-Navigator; it might only work with a future release version of TI-Navigator.

Computer HubNet

NOTE: Computer HubNet is still in the process of being developed and is only being released as a beta version. There may be some features that are lacking or bugs that we have not found. However, we believe that it is robust enough to use unassisted. Please send us any feature requests or bugs that you find to feedback@ccl.northwestern.edu

- **A networked computer with NetLogo installed and, optionally, an attached projector for the leader.** This computer will run NetLogo and act as the server for the HubNet Clients. The projector, if attached, will project the simulation for class viewing.

NOTE: By default, the NetLogo Graphics Window is not sent to the HubNet Clients since we have found that with large numbers of clients this slows down the simulation. For instructions on how to make a HubNet model send the NetLogo Graphics Window to clients, please refer to [the Computer HubNet portion of the Running an activity](#).

- **A networked computer with NetLogo installed for each participant.**

First-time NetLogo user?

NetLogo is a programmable modeling environment. It comes with a large library of existing simulations, both participatory and traditional, that you can use and modify. Content areas include social science and economics, biology and medicine, physics and chemistry, and mathematics and computer science. You and your students can also use it to build your own simulations, if you choose.

In traditional NetLogo simulations, the simulation runs according to rules that the simulation author specifies. HubNet adds a new dimension to NetLogo by letting simulations run not just according to rules, but by direct human participation. Since HubNet builds upon NetLogo, we recommend that before trying HubNet for the first time, you should be familiar with the basics of NetLogo.

Teacher workshops

For information on upcoming workshops and NetLogo and HubNet use in the classroom, please contact us at feedback@ccl.northwestern.edu.

Getting Started With HubNet

Using NetLogo

We recommend that you become familiar with NetLogo itself before using the HubNet technology.

You can become familiar with NetLogo by trying out some of the models in the Models Library. Open the Models Library from the File menu in NetLogo. Then click on a model that you want to try and press the Open button. The Information tab in each of the models gives background information and instructions.

Other sections of the NetLogo User Manual may be helpful when learning NetLogo. We suggest that beginning users focus on the section [Tutorial #1: Running Models](#).

If you have any questions about NetLogo, feel free to E-mail us. You can reach us at feedback@ccl.northwestern.edu.

HubNet Activities

Below are the current HubNet activities that are fully developed. Be aware that some of these models have only been implemented for one type of HubNet. For many models, you will find its educational goals and suggested ways to incorporate them into your classroom in the Participatory Simulations Guide which can be found on the [Participatory Simulations Project web site](#).

- Disease A disease spreads through the simulated population of students.
- Elevators Student-controlled elevators demonstrate the relation of velocity and position.
- Function Activity Students experience the concept of a function by forming themselves into graphs.
- Gridlock Students use traffic lights to control the flow of traffic through a city.
- People Molecules Using CBRs (Calculator Based Range-finders), students use their bodies to represent gas molecules.
- Polling Ask students questions and plot their answers.
- Regression As students move on-screen, they see the best-fit line of their positions.

NOTE: In addition to the discussion of learning goals and classroom techniques, these materials also contain step by step instructions and screen shots. As of December 2002, they are out of date and no longer match the actual activities in many respects. We are working on updating them. In the meantime, please use these materials for the discussions, but for step by step instructions, rely instead on the QuickStart Instructions built into the activities (see next section).

Running an activity

You'll find the HubNet activities in NetLogo's Models Library, under the HubNet Calculator Activities and HubNet Computer Activities folders.

In each of the activities, you'll see a box on the screen labeled QuickStart Instructions. It contains step by step instructions on how to run that activity. Click the Next>>> button to advance to the next instruction.

We suggest doing a few practice runs of the activity before trying it in front of an actual class.

If you have any questions about running the activities, feel free to E-mail us. You can reach us at feedback@ccl.northwestern.edu.

Calculator HubNet

When you open the first Calculator HubNet activity for each session of NetLogo, you will be prompted by the TI-Navigator Login dialog. This prompts you to enter information (such as User Id or Password) that is necessary for connecting to the TI-Navigator system and running the HubNet activity. If you don't actually want to run the model, just press the Cancel button.

For more information about how to log in to the calculators and other details of using Calculator HubNet, please refer to the Participatory Simulations Guide which can be found on the [Participatory Simulations Project web site](#).

Computer HubNet

Setting up a Computer Activity

Opening a Computer HubNet model will cause NetLogo to start a server which will allow people to join the activity. You should then follow the instructions for the particular model found in the QuickStart Instructions monitor. Most models will require you to press a forever button, often called GO. You, as the leader, should notify everyone that they may join. For them to join, you should give them the IP address for the computer that is running NetLogo. The IP address of the server can be found in the HubNet Control Center, which can be opened by choosing the HubNet Control Center option in NetLogos Tools menu. Every user will have to enter this IP address to be able to enter the activity. On some systems, the HubNet Client will automatically detect all the Computer HubNet activities currently being run. On these systems, instead of typing in the IP address, everyone can select the activity from a list. The list should contain the name of the activity followed by the IP address of the computer running NetLogo.

Every participant will also have to enter a unique User Id. A good method for choosing a unique User Id is to combine the first and last name of the user. For example, if the users name is Joe Smith, he might want to choose the User Id, joesmith. Another good method for getting a unique User Id is for you, the leader, give out unique User Ids to everyone. If you decide to use this second method, be sure to inform every user what User Id he or she is supposed to use.

Joining a Computer Activity

Once the leader has informed you that you can join, you should open up the HubNet Client application. This will prompt you for the IP address of the computer running NetLogo. Enter the IP address that the leader gave you. Alternately, if your system supports the Computer HubNet discovery feature, you may instead choose the activity from the list shown.

Regardless of how you choose the Computer HubNet server, you will also have to enter a User Id. Please enter one. The leader will give directions on how to choose one. If no one else is using the

one you enter, NetLogo will send you the client interface and you can start playing. If you didn't choose a unique User Id, your client application will display a message indicating this. You should keep trying new User Ids until you are able to enter. A good method for choosing a unique User Id is to use your first and last name together. For example, if your name is Joe Smith, you might want to choose the User Id, joesmith.

HubNet Control Center

One feature of Computer HubNet is the HubNet Control Center. You can open this using the HubNet Control Center option in the Tools Menu when a Computer HubNet Activity is open and the server is running. The Control Center shows you useful information, such as the IP address of the computer, who is logged in, etc. It also allows you to disconnect clients from the activity and send out messages for all the clients to read.

In addition to these features, the HubNet Control Center allows you to control whether debugging output is on or off. Debugging output is useful when creating your own HubNet activity or to keep track of which participant sent what data when. Any output is sent to the log.txt file on Windows machines, or the Console to Macs. The Control Center also allows you to control whether the NetLogo Graphics Window and plots are mirrored on the clients. These options are useful if there are a lot of changes to the Graphics Window or plots or if you just don't wish the clients to see what is happening on the clients. By default, these options are turned off since they are in an experimental stage.

HubNet Authoring Guide

If you wish to learn more about authoring or modifying HubNet activities, you should look at the [HubNet Authoring Guide](#).

Computer HubNet Troubleshooting Tips

Here are some things that have been known to go wrong and ways of fixing or working around them.

I have tried quitting a HubNet Client, but it just sits there and won't quit no matter how many times I try.

You will have to force the client to quit. On Macs, force quit the application (Command+Option+Esc). On Windows, open the Windows Task Manager (Ctrl+Alt+Delete), select Hubnet and press End Task. Then you should inform the leader of the simulation to use the HubNet Control Center to kick your the client out of the simulation.

We believe that we have resolved all instances of these problems with the exception of the client trying to quit when it has lost its network connection. If you encounter this problem and still have a network connection, please [let us know](#).

When I open a Computer HubNet Activity, I got a dialog that said:

Could not start the Computer HubNet server. Another program may be using the port that is needed.

Another program (perhaps another copy of NetLogo) is using a port or some other resource that HubNet needs. Check to see if you have another copy of NetLogo open that is running Computer HubNet. If you do, quit it and then try reopening the model. If you don't, you can try quitting other programs and reopening the model until you are able to start the Computer HubNet server successfully.

Somehow two or more HubNet Clients are controlling the same agents in the simulation and they aren't supposed to. What can I do to fix this problem?

Remove all but one of the clients from the simulation by either by having the people exit the activity or by using the Kick Client feature in the HubNet Control Center to forcibly remove them. If you wish them all to still be in the simulation, the ones that quit should log back in with a different user name.

I had run and then stopped a HubNet model. I changed windows or minimized NetLogo or did something else to obscure or redraw the Graphics Window. When I came back to the NetLogo, the Graphics Window was all gray. What happened? Is all my data lost?

In order to reduce the number of messages sent to the clients, we use the `no-display` and `display` primitives to turn off the drawing to the Graphics Window and only update it periodically. This has the unfortunate side effect of causing the Graphics Window to become gray if it is redrawn and the GO button is not pressed. You can simply type `display` into the command center to redraw the current state of the turtles and patches in this situation.

My computer went to sleep while running a HubNet activity. When I woke the computer up, I got an error and HubNet wouldn't work anymore.

We have reports that if a computer goes to sleep while running Computer HubNet (either the HubNet Client or the server), sometimes things do not work after the computer wakes up. Please send us the error message along with the situation that caused it. However, we suggest that people keep their computers from sleeping while running an activity.

I'm trying to test a HubNet activity with both the server and clients on my Mac, without a network, but my clients can't seem to connect to the server.

On a machine running Mac OS 9, you need to connect to a network, at least temporarily, in order for this to work. Operating systems provide a special network loopback (usually called localhost), to allow programs to connect to the machine on which they're running, even if that machine isn't networked. Mac OS 9, however, doesn't have a localhost until you connect to a network, at which point it is activated.

If your situation does not fall into one of the above problem descriptions, please [send us a bug report](#).

Known Computer HubNet Issues

If HubNet malfunctions, please send us a bug report. See the [Contact Us](#) section for instructions.

Known bugs (all systems)

- HubNet has not yet been extensively tested with large numbers of clients (i.e. more than about 20). Unexpected results may occur with more clients.
- Increasing the number of clients can increase the response time for clients.
- There are situations that require the clients to force quit the client application. For example:
 - ◆ If the network connection is lost by the clients or the server, even just for a few minutes, the clients and/or the server might need to be forcibly quit.
- Out-of-memory conditions are not handled gracefully
- Graphics updates can take a long time.
- Sending large amounts of plotting messages to the clients can take a long time.
- NetLogo may pause from time to time during a run.
- NetLogo does not handle malicious clients in a robust manner (in other words, it is likely vulnerable to denial-of-service type attacks).
- Performance does not degrade gracefully over slow or unreliable network connections.
- If you are on a wireless network or sub-LAN, the IP address in the HubNet Control Center is not always the entire IP address of the server.
- Authoring new HubNet activities is more arcane and difficult than it should be.
- HubNet.jar is significantly larger than it should be.
- There are a number of known situations in which Java exceptions can result during communication with clients.
- The focus of the cursor sometimes is not in the correct location when logging in on a client.

Shapes Editor Guide

The Shapes Editor allows you to create and save turtle designs. NetLogo uses fully scalable and rotatable vector graphics, which means it lets you create designs by combining basic geometric figures, which can appear on-screen in any size or orientation.

Getting Started

To begin making shapes, choose **Shapes Editor** in the Tools menu. A new window will open listing all the shapes currently in the model, beginning with *default*, the default shape. The Shapes Editor allows you to create a new shape, edit, copy, or delete an existing shape, or import shapes from other models.

Creating and Editing Shapes

Pressing the **New** button, will cause a new shape to be created and the editing window will appear, allowing you to name and make the shape. The shape you make will appear in the main drawing area and in the three smaller preview areas found on the left side of the editing window.

The preview areas show your shape at different sizes as it might appear within your model, as well as how it looks while rotating. The rotatable feature can be turned off if you want a shape that does not rotate as the turtles heading changes.

There are four drawing tools (select from four button icons), each of which can be filled with color or not, and a background grid to guide you. You can move and size these basic shapes with the mouse pointer as a drawing tool.

The basic geometric figure created last will fall on top. Geometric figures can be removed one at a time with the **Remove Last** button. The **Remove Last** and **Remove All** buttons are the only way of correcting mistakes you can't go back to move or alter a geometric figure after it's been drawn. If you make a mistake drawing a geometric figure, to fix it, remove it and then redraw it.

Geometric figures that use the *key color* (selected from a drop-down menu the default is gray) will change according to the value of each turtles *color* variable in your model. Figures filled with any other color (selected from the palette at the top) will stay that color regardless of each turtles *color*. For example, you could create cars that always have yellow headlights and black wheels but have bodies of different colors by drawing the bodies in the *key color*.

It's tempting to draw complicated, interesting shapes, but remember that in most models, the patch size is so small that you won't be able to see all the detail. Simple, bold shapes are best.

When the shape is done, give it a name and press the **Done** button at the bottom of the editing window. The shape and its name will now be included in the Shapes Editors list along with the default shape.

If you want to use a shape from another model in this model, you must first import the shape into this model. Press the **Import** button to select a NetLogo model from which to import one or more shapes. Once you have chosen a model, a list of that model's shapes will appear. Choose as many of these shapes as you like to import into the original model and press the **Import** button to import

the shapes.

Using Shapes in a Model

In the models code or in the command center, you can use any of the shapes that are in the model. For example, suppose you want to create 50 turtles with the shape rabbit. Provided there is some shape called *rabbit* in this model, give this command to the observer in the command center:

```
crt 50
```

And then give these commands to the turtles to spread them out, then change their shape:

```
fd random 15  
set shape "rabbit"
```

Voila! Rabbits! Note the use of quotation marks around the shape name.

The `set-default-shape` primitive is also useful for assigning shapes to turtles.

BehaviorSpace Guide

This guide is broken up into three parts:

- [What is BehaviorSpace?](#): A general description of the tool, including the ideas and principles behind it.
- [How it Works](#): Walks you through how to use the tool and highlights its most commonly used features.
- [Interface Reference](#): Offers comprehensive descriptions of each main window, as well as the interface components within those windows.

What is BehaviorSpace?

BehaviorSpace is a software tool integrated with NetLogo that allows you to perform experiments with models. It systematically varying the values of sliders and records the results of each corresponding model run. This way you can explore the models space of possible behaviors and determine which combinations of slider values cause the behaviors of interest.

The need for this type of experiment is revealed by the following observations: Models often have more than one slider, each of which can take a large range of values. Together they form what in mathematics is called a parameter space for the model, whose dimensions are the number of sliders, and in which every point is a particular combination of values. Running a model with different slider values (and sometimes even the same ones) can lead to drastically different behavior in the system being modeled. So, how are you to know which particular configuration of values, or types of configurations, will yield the kind of behavior you are interested in? This amounts to the question of where in its huge, multi-dimension parameter space does your model perform best?

Suppose you want speedy synchronization from the agents in the Fireflies model. The model has four sliders number, cycle-length, flash-length and number-flashes that have approximately 2000, 100, 10 and 3 possible values, respectively. That means there are $2000 * 100 * 10 * 3 = 600,000$ possible combinations of slider values! Trying combinations one at a time is hardly an efficient way to learn which one will evoke the speediest synchronization.

BehaviorSpace offers you a much better way to solve this problem. If you specify a subset of values from the ranges of each slider, it will run the model with each possible combination of those values and, during each model run, record the results. In doing so, it samples the models parameter space not exhaustively, but enough so that you will be able to see relationships form between different sliders and the behavior of the system. Once all the runs are over, you are presented with a visual display with which you can explore the data from all the runs and determine how well each exhibited the behavior you are investigating.

The idea behind BehaviorSpace is that the way to truly understand a model is to run it multiple times with different parameter (slider) settings in order to see the whole range of behaviors the system is capable of producing. Only then is it possible to investigate when and why certain behaviors arise. Isolated trials are insufficient for this purpose because you have no reason to assume that the model will always demonstrate the particular behaviors you see. Its like eating in one restaurant in New York and then claiming youve seen all that the city has to offer. By enabling you to explore the entire parameter space of a model, and thus its entire space of behaviors,

BehaviorSpace can be a powerful tool for model understanding.

How It Works

You begin using BehaviorSpace by setting up an experiment. Its purpose is to run the model many times and collect data to determine which run best exhibits the behavior you are after. In the case of Fireflies, as mentioned above, data might be collected to reveal how quickly synchronization happened in each run.

Setting up an experiment initially entails specifying in the **slider values dialog** what possible values each slider in the model should take in the experiment, either in the form of a range or an arbitrary list, which is then displayed in the **sliders list**.

You can also dictate how long each run should last, either by entering the number of times the forever button should be called in the **time clicks text-box**, or by providing a reporter in the **exit condition text-area** that causes the run to stop when it reports a boolean value of true.

Next, you must specify a **behavior reporter**, which is the data you want continually recorded during a model run. For Fireflies, this would be the current extent of synchronization in the model, say the number of fireflies currently lit up, which you could enter with the following reporter:

```
count turtles with [color = yellow]
```

By collecting this data repeatedly throughout a run, an overall assessment can later be made as to how quickly synchronization was achieved.

Finally, you may select a **setup button** to initialize the model and the **forever button** that continues it.

By inputting the above information, you have told the system enough that it can run an experiment. As the experiment proceeds, it will produce data about each model run based on the behavior reporter you gave it.

In its raw form, this data is dry and difficult to interpret it is just long series of numbers. BehaviorSpaces job makes the data intuitively accessible by presenting it visually. It does so by allowing you to hold the values of all sliders but two constant, then presenting you with a grid of squares, called a **fitness landscape**.

The axes of the fitness landscape represent the two selected sliders. Each square in the landscape then represents the particular run whose slider values are determined by the location of the square along the two axis and the current values of the sliders held constant.

Example: Suppose you are performing an experiment with the Fireflies model that tried many possible combinations of slider values. Then you select flash-length and number-flashes as the sliders to represent the axes of the landscape, with number held constant at 1500 and cycle-length at 100. Assuming flash-length is on the horizontal axis and number-flashes is on the vertical axis, the landscape would be 10 by 3, since flash-length goes from 1 to 10 and number-flashes from 1 to 3. Each square on the landscape would be colored a particular shade of blue, ranging from

solid blue to black, depending on how good (i.e. fit) the behavior of the run represented by that square was according to criteria you select. For instance, squares representing runs of Fireflies in which the agents synchronized their lights quickly would be more blue than those representing runs where the fireflies synchronized slowly or not at all.

The reason the grid is called the fitness landscape is because it displays a 2-dimensional landscape of squares, each with a fitness, allowing you to take in with a glance regions of high and low fitness.

The criteria with which you rank run data and thereby determine fitness can be based on simple **point statistics**, such as maximum or mean value, or by **slope**, which is sensitive to the data's behavior (not to be confused with the model's behavior) over time.

In addition to changing the values of the constant sliders and alternating which sliders get to be on the axes of the landscape (these two operations let you explore the model's parameter space), you can see the data from each run represented on the landscape plotted on a **behavior plot** by moving your mouse over that run's corresponding square.

You can learn more about how evaluation of the experimental data works in the evaluation window part of the Interface Reference section below.

Interface Reference

This section describes the interface to BehaviorSpace. Included is a general description of each of the three main BehaviorSpace windows, a list of the interface components they contain, as well as a description of those components. It is intended mainly as a reference. The windows are discussed in the order you use them: first the Setup Window, then the Progress Dialog, and finally the Evaluation Window.

Setup Window

This is the window in which you set up an experiment. Setting up an experiment requires entering information in the fields outlined below, then pushing the **run comparison button**. When opening and closing models, or changing the existing model, the setup window is automatically updated, so there is no need to close and reopen it.

Component descriptions

Sliders list: The current list of sliders in the model, which is updated automatically as sliders are added, changed or deleted. The values specified to the right of each slider are the current values that slider is assigned to take in the experiment.

Change values button: Brings up the **Slider values dialog** so you can change slider values.

Slider values dialog: The dialog that pops up when the **change values button** is pushed.

- To specify values in a list, enter the values in the upper text-field. Values in the list can be separated by one or more spaces, commas, tabs, or semicolons.

- To specify values in a range, enter the first and last values the slider should take and the increment by which to traverse this range.
- First and last values of a range are inclusive.
- The first value must be less than or equal to the last one.
- The increment must be greater than zero.
- If a single value is entered more than once, then each instance will be treated as a separate value that is, if a particular slider has four 5s, then that slider will have the value 5 four times.
- To leave the dialog, there must be valid values in both the list and range text-fields.
- The values used for a particular slider in an experiment will be those whose checkbox is currently selected. For example, if the values range checkbox is selected, then the values specified by the range will be used, not those in the list.

Behavior reporter text area: A place to enter a NetLogo reporter to specify what kind of data is to be collected during each model run. This reporter is evaluated before every time click, or call to the forever button, and once at the end of the run, and its value is recorded for later analysis.

- Input must be a valid NetLogo reporter.
- Can use any reporters, whether built into the NetLogo language or written in the model.
- It must report a number.

Example: If you are interested in the population of sick people in the Virus model, you might want your behavior reporter to report how many sick people there currently are in the model with the following behavior reporter:

```
count people with [sick?]
```

Or, if you are investigating the distance between each turtle and the center of the screen in Turtles Circling, you might have it report the mean of all the turtles distances to the center:

```
mean values-from turtles [distancexy 0 0]
```

Time clicks checkbox and text-field: When the checkbox is selected, each model run will end after the number of time clicks specified in the text-field. (One time click means one push, or iteration, of the forever button.)

- Input to the text-field must be a valid integer, *not* just any numeric reporter.

Exit condition checkbox and text-field: When the checkbox is selected, each model run will continue until the reporter entered in the text-field reports true. This reporter is run after every time click. The exit condition can be any valid NetLogo reporter that evaluates to true or false, such as those that use the operators =, <, >, <=, or >=

Example: The following exit condition causes model runs of Wolf-Sheep Predation to stop after either population is extinct:

```
(not any sheep) or (not any wolves)
```

Setup and forever button menus: Allow you to select the optional setup button and mandatory

forever button for an experiment. The setup button, if selected, is called once at the beginning of a model run, while the forever button is called repeatedly to run the model.

- To forgo the use of a setup button in an experiment, select the <none> option in the setup button menu.
- If you choose, the same button can be used as both the setup and forever buttons.
- If either the setup or forever button is chosen to be a forever button in your model, then one call to it (i.e. one time click) amounts to one call to its code, *not* to repeated calling of that code, just as if the actual button had been pressed.

Run comparison button: Begins the experiment, which is based on the input to the interface elements listed above.

Progress Dialog

This dialog appears automatically when an experiment begins to show you its progress. It contains real-time information about the current model run and the experiment as a whole. (It isn't necessary that you always watch model runs as they progress, but it may be helpful to monitor the progress of the experiment.) The dialog can be exited safely by hitting the **cancel button**. This will abort the experiment. If the experiment is allowed to run to completion, then this dialog disappears and is replaced by the Evaluation Window.

Component descriptions

Graphics window: Displays the state of the current model run.

- Has a fixed size, and does not allow for more than 300 patches across in either direction
- Is completely equivalent to the graphics window in the main interface in all other respects

Graphics on? checkbox: When unselected, graphics are temporarily turned off, although the model proceeds normally.

- Turning graphics off can make your experiment run significantly faster
- Can be selected and unselected repeatedly during an experiment

Current run number / slider values / time click text: Displays the number of the current run being performed, the values of the sliders in that run, and the number of time clicks that have elapsed

Behavior plot: Plots the value of the **behavior reporter** as the model run progresses.

Cancel button: Aborts the experiment, at which point the **progress dialog** quits.

Evaluation Window

The evaluation window is perhaps the most complex part of BehaviorSpace, but also the most flexible and powerful. It is designed to allow you to visually explore the behavior data reported by the model runs.

It works by holding the values of all but two sliders constant, then drawing a grid of squares, called a fitness landscape, that represents all possible value combinations of the last two. The values held

constant by the former sliders can be changed, and any of these sliders can be swapped in for one of the two sliders representing the axes on the grid. In this way you are essentially focusing on two sliders at a time, seeing what relationships exist between their values and the behavior of the model.

At the top of the window is a collection of radio boxes and pull-down menus, through which you can select criteria to rank the data reported by the different runs. These criteria can be changed as often as you wish, enabling you to highlight more than just one property of the data. Data can be ranked by slope characteristics, such as how much it increases over time, or the extent to which it can be fit to an exponential curve; or, the data can be ranked by a point statistic, such as its mean or minimum value.

Finally, you are able to mouse-over the landscape, and as the mouse passes over each square, which represents a single model run, the data from that run is plotted on the behavior plot to the left. Every time an experiment is run, a new evaluation window is created, so there can potentially be many on the screen at once. Each, however, only represents the data from the experiment that created it, so even if you change the name or number of sliders in your model, existing windows will remain unchanged.

Component descriptions

Point statistic checkbox: When this is selected, the data obtained from each run will be ranked according to the items selected by the following two menus. All three slope menus are made irrelevant. In general, it means that the data will be judged based on a simple, global statistic, instead of how it varies over time.

Point statistic type menu: Determines whether run data should be ranked according to their minimum, maximum, mean, or last value, or by the number of time clicks they lasted for. The latter option is only relevant when an exit condition was entered to stop runs, otherwise they will all finish in the same number of time clicks.

Point statistic value menu: Determines whether it is better for the statistic selected in the **point statistic type menu** to be high, low, or near 0.

- Low means more negative, so -100 is lower than 1
- Near 0 means as close to 0 as possible, irrespective of sign, so -0.5 is nearer to 0 than 1

Slope checkbox: When selected, the data from each run will be ranked according to how it behaves over time, as specified by the following three menus. The items selected by the previous two menus are then irrelevant.

Slope type menu: Controls whether the run data is fit to a line or exponential curve. This distinction is only relevant with regards to error, since the slope of the data always remains the same. The higher the weight selected in the **relative weight menu**, the less this menu selection matters, because that means slope matters increasingly more than error. Error is determined by attempting to fit the data to either a line or a curve, depending on which is chosen, and then finding the Least-Square Error involved in doing so.

- Fitting data to a line means finding a function of the form $y = ax + b$ that best approximates the data.

- Fitting data to an exponential curve means finding a function of the form $y = ae^{(bx)}$, where $e \approx 2.71828$, that best approximates the data.
- If exponential is selected and any run has both positive and negative data values, then a warning dialog is brought up saying that the data cannot be fit to an exponential curve, and linear is automatically selected. This occurs because exponential curves of this form can be either above the x-axis (when a is positive) or below the x-axis (when a is negative), but not both.

Slope value menu: Determines whether it is better for the data to be increasing (the more positive the slope the better), decreasing (the more negative the better), or constant (the closer to 0 the better)

Relative weight menu: Determines the relative importance of the slope of the data compared to how well it fits a line or exponential curve (see **slope type menu**). If 60 is chosen, then 60% of the data's rank comes from its slope (how much it is increasing, decreasing or constant) and 40% comes from line and curve-fitting error. So, a value of 100 means only slope is considered, while a value of 0 means only the error is considered.

- If you do not care about seeing strictly linear or exponential behavior in the run data, you should use high relative weight values, and otherwise use low ones.

Fitness landscape: Consists of a grid of squares, each of which represents the model run whose slider values were the current values of the constant sliders below and the i th and j th values of the two sliders on the grid's axes, where i is the horizontal index of the square and j is the vertical. The entire grid thus represents *all* runs in which the constant sliders had their current values (the current values being the ones displayed on the **set of slider components held constant** below the grid). This grid earns the name fitness landscape because the squares of which it is composed each has a fitness value, expressed by the color of the square, enabling you to see a landscape of colors denoting regions of high and low fitness—that is, sets of neighboring near-black and near-blue squares. The landscape represents a 2-dimensional plane in the multi-dimensional parameter space of the model, since it accounts for all possible values of two sliders, while the rest are held constant. By changing the values of any constant sliders you are shifting that 2-D plane through the parameter space, and by selecting new sliders to be the axes, you are selecting a new plane entirely (that is, a plane through different dimensions).

- As you move your mouse over different squares in the grid, the data from those runs will appear in the **behavior plot** to the left, in addition to the line or curve that best fits the data if the **add best-fit-line checkbox** is selected.
- As your mouse passes over a square in the landscape, information about the model run that square represents appears above the landscape. This information consists of the value that each of the two selected sliders had during that run, and the overall fitness of that run.
- The fitness of a run is an estimation of how good the run was according to the criteria you select in the menus listed above. When the **slope checkbox** is selected, fitness derives from the slope of the data as well as how well it fits a linear or exponential curve. You can control the relative importance of these factors with the **relative weight menu**. When the **point statistic checkbox** is selected, the value that you see above the landscape is not the fitness of the run, but the value of the chosen statistic. For instance, the minimum value of the run is shown when minimum value is selected in the **point statistic type menu**. This value is *not* necessarily the fitness of the run, however, since the goodness of a particular value changes when you make a new selection in the **point statistic value menu**.

- The grid has a maximum height and width, which is reached when there are more than 5 squares in either direction, and cannot be resized.
- Squares turn gray when there is insufficient information to construct a grid this happens when either axis has not been assigned a slider (one axis is sufficient if a model only has one slider).

Set of sliders components held constant: These sliders consist of all those not present as axes of the **fitness landscape**. Since all the model runs represented on the landscape have the same value for these sliders namely, the one currently selected on the slider components they are considered held constant for the landscape. By changing the value of a single one of these sliders the whole landscape changes, because it now represents a whole new set of runs. If you changed a slider from 10 to 12, say, then all the runs in which that slider had the value 10 are replaced by those in which it had the value 12.

- Changing the value of these sliders does *not* affect the value of the sliders on the main interface.
- The only values the sliders can take are those that they actually had in the experiment.

Behavior plot: Displays the data from individual runs as you move the mouse over their corresponding squares on the **fitness landscape**. Its x-axis is time clicks and its y-axis is behavior that is, the value reported by the **behavior reporter** entered in the setup window.

- The best fit line or curve can be added along with the data if the **add best-fit-line checkbox** is selected.
- Run data can be continually collected in the plot and superimposed on each other by selecting the **superimpose plots checkbox**.
- Only 14 plot pen colors are used, so if more than 14 data sets are superimposed at a single time, the system will begin reusing colors for the new pens.
- Automatically scales to show all data.

Add best-fit-line checkbox: Adds the line or exponential curve that best fits the data to the **behavior plot**, depending on whether linear or exponential was selected in the **slope type menu**.

Superimpose plots checkbox: Stops old data from being removed from the **behavior plot** when new data is added, resulting in increasing sets of data being superimposed on each other. Unselected, the checkbox causes all data to be removed.

Export Plot button: Saves the data currently being displayed in the **behavior plot** to a file.

Export Behavior Data button: Saves the behavior data gathered during all of the model runs to a file. The data saved for each model run includes: the slider settings for that run; how many time ticks the run lasted; the minimum, maximum, average, and final value for the behavior reporter; and the value of the behavior reporter at each time tick during the run. (Note that the fitnesses are not exported, only the raw behavior data. In a future version of BehaviorSpace the fitness data will be exported as well.)

Note: The export buttons create files in plain-text, comma-separated values (.csv) format. CSV files can be read by most popular spreadsheet and database programs as well as any text editor.

FAQ (Frequently Asked Questions)

Feedback from users is very valuable to us in designing and improving NetLogo. We'd like to hear from you. Please send comments, suggestions, and questions to feedback@ccl.northwestern.edu, and bug reports to bugs@ccl.northwestern.edu.

General

- [What is NetLogo written in?](#)
- [How do I cite NetLogo in an academic publication?](#)
- [How do I cite a model from the Models Library in an academic publication?](#)
- [What license is NetLogo released under? Are there any legal restrictions on use, redistribution, etc.?](#)
- [Is the source code to NetLogo available?](#)
- [Do you offer any workshops or other training opportunities for NetLogo?](#)
- [What's the difference between StarLogo, MacStarLogo, StarLogoT, and NetLogo?](#)
- [Has anyone written a model of <x>?](#)
- [Are NetLogo models runs scientifically reproducible?](#)

Downloading

- [The download form doesn't work for me. Can I have a direct link to the software?](#)
- [Downloading NetLogo takes too long. Is it available any other way, such as on a CD?](#)
- [I downloaded and installed NetLogo but the Models Library has few or no models in it. How can I fix this?](#)
- [Can I have multiple versions of NetLogo installed at the same time?](#)

Usage

- [How do I change the number of patches?](#)
- [How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?](#)
- [Can I import a graphic into NetLogo?](#)
- [My model runs slowly. How can I speed it up?](#)
- [I want to try HubNet. Can I?](#)
- [Can I run a NetLogo model from the command line? Can I run it without a GUI?](#)
- [Can I have more than one model open at a time?](#)
- [Can I copy or save a picture of the graphics window?](#)
- [Can I make a movie of my model?](#)

Programming

- [How is the NetLogo language different from the StarLogoT language? How do I convert my StarLogoT model to NetLogo?](#)
- [The NetLogo world is a torus, that is, the edges of the screen are connected to each other, so turtles and patches wrap around. Can I use a different world topology: bounded, infinite plane, sphere, etc.?](#)
- [Does NetLogo have a command like StarLogo's grab command?](#)
- [I tried to put `-at` after the name of a variable, for example `variable-at -1 0`, but NetLogo won't let me. Why not?](#)

- [I'm getting numbers like 0.10000000004 and 0.79999999999 instead of 0.1 and 0.8. Why?](#)
- [Can I sort a list according to a sort order I define myself?](#)
- [Is there some simple way to multiply one list by another list or number without writing a procedure?](#)
- [Can I read data from a file?](#)
- [Can I write data to a file?](#)
- [How can I use different patch neighborhoods \(circular, Von Neumann, Moore, etc.\)?](#)
- [Can I connect turtles with lines, to indicate connections between them?](#)
- [How can I keep two turtles from occupying the same patch?](#)
- [How does NetLogo decide when to switch from agent to agent when running code?](#)

General

What is NetLogo written in?

NetLogo is written entirely in Java (version 1.1).

How do I cite NetLogo in an academic publication?

NetLogo itself: Wilensky, U. 1999. NetLogo. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer–Based Modeling, Northwestern University. Evanston, IL.

HubNet: Wilensky, U. Stroup, W., 1999. HubNet. <http://ccl.northwestern.edu/ps/>. Center for Connected Learning and Computer–Based Modeling, Northwestern University. Evanston, IL.

How do I cite a model from the Models Library in an academic publication?

Wilensky, U. (year). Name of Model. URL of model. Center for Connected Learning and Computer–Based Modeling, Northwestern University. Evanston, IL.

To determine the URL for a model, visit [our web–based version of the Models Library](#) and click on the name of the model. An example model URL is:

<http://ccl.northwestern.edu/netlogo/models/PepperedMoths>.

To determine the year, open the model from the NetLogo application and look in the copyright information at the bottom of the Procedures tab.

What license is NetLogo released under? Are there any legal restrictions on use, redistribution, etc.?

The NetLogo software, models and documentation are distributed free of charge for use by the public to explore and construct models. Permission to copy or modify the NetLogo software, models and documentation for educational and research purposes only and without fee is hereby granted, provided that this copyright notice and the original author's name appears on all copies and supporting documentation. For any other uses of this software, in original or modified form, including but not limited to distribution in whole or in part, specific prior permission must be obtained from Uri Wilensky. The software, models and documentation shall not be used, rewritten, or adapted as the basis of a commercial software or hardware product without first obtaining appropriate licenses from Uri Wilensky. We make no representations about the suitability of this software for any purpose. It is

provided as is without express or implied warranty.

We are in the process of reevaluating the language of the license in response to user feedback. In the future, we intend to send out a revised license.

Is the source code to NetLogo available?

At present, no. We are evaluating how best to distribute NetLogo when it is in a more mature state. Making the source available is one possibility.

We do understand, however, that it is important that NetLogo not be a closed and non-extensible platform. That is not our intention for the product. For a future version of NetLogo, we plan to add an API so that users can add their own extensions to NetLogo. We'd like to work with interested users to help design this API to suit the needs of our user community.

Do you offer any workshops or other training opportunities for NetLogo?

We offer workshops from time to time. If a workshop has been scheduled, we will announce it on the NetLogo home page and on the netlogo-users group. If interested in this type of opportunity, please contact us at feedback@ccl.northwestern.edu if interested.

You should also check the NetLogo User Community Models section of our [Models Library](#) web page.

What's the difference between StarLogo, MacStarLogo, StarLogoT, and NetLogo?

The first version of StarLogo was developed at the MIT Media Lab in 1989/90. This original version ran on a massively parallel supercomputer called The Connection Machine. A few years later (1994), a simulated parallel version was developed for the Macintosh computer. That version eventually became MacStarLogo. The [StarLogoT](#) (1997) software, developed at the Center for Connected Learning and Computer-Based Modeling (CCL), is essentially an extended version of MacStarLogo with many additional features and capabilities.

Since then two multi-platform Java-based multi-agent Logos have been developed: NetLogo (from the CCL) and a Java-based version of StarLogo (from MIT).

The NetLogo language and environment differ in many respects from MIT StarLogo's. Both languages were inspired by the original StarLogo, but were redesigned in different ways. NetLogo's design was driven by the need to revise and expand the language so it is easier to use and more powerful, and by the need to support the HubNet architecture. NetLogo also incorporates almost all of the extended functionality of our earlier StarLogoT. See [below](#) for information on how to convert a StarLogoT model to NetLogo.

You can use the chart below to help familiarize yourself with the new features NetLogo has to offer.

StarLogoT	NetLogo	Features
X	X	Fully programmable
X	X	Language is Logo dialect extended to support agents and parallelism
X	X	Interface builder w/ buttons, sliders, switches, choices, monitors, and text

		boxes
X	X	Info area for annotating your model
X	X	Powerful and flexible plotting system
X	X	Agent Monitors for inspecting agents
X	X	Export and import model function (save and restore state of model)
	X	Cross-platform: runs on MacOS, Windows, Linux, et al
	X	Models can be saved as applets to be embedded in a web page
	X	Unlimited numbers of agents and variables
	X	Double precision arithmetic
	X	Simplified language structure
	X	Agentsets make many programming tasks easier
	X	Syntax-highlighting code editor
	X	Rotatable and scalable vector turtle shapes
	X	Exact on-screen turtle positioning
	X	Redesigned user interface
	X	Text labels for turtles and patches
	X	New primitives
	X	BehaviorSpace: a tool used to collect data from multiple runs of a model
	X	HubNet: participatory simulations using networked devices
	X	Converts StarLogoT models into NetLogo models

Has anyone written a model of <x>?

The best place to ask this question is on the [NetLogo Users Group](#).

Are NetLogo models runs scientifically reproducible?

Yes, with the following cautions:

- If your model uses random numbers, then in order to get reproducible behavior, you must use the `random-seed` command to set the random seed, so that your model will receive the exact same sequence of random numbers every time.
- If your model uses the `every` or `wait` commands in such a way that affects the outcome of the model, then you may get different results on different computers, or even on the same computer, since the model may run at a different speed. (Such models are rare these two commands are common, but using them in a way that affects the outcome is not.)
- In order to reproduce model runs exactly, you must be using the exact same version of NetLogo. The details of the agent scheduling mechanism and the random number generator may change between NetLogo versions, and other changes (bugfixes in the engine, language changes, and so forth) may also affect the behavior of your model. (Then again, they may not.)
- NetLogo is written in Java, so the exact reproducibility of results is only as good as the underlying Java implementation. Java 1.1 does not guarantee that the result of mathematical operations will be bit-for-bit identical from platform to platform. Any differences will of course be very small, but thanks to the butterfly effect small numerical differences can

eventually be magnified by some models into large differences in behavior. Starting in Java 1.3, Sun has provided an alternate math library that does guarantee bit-for-bit identical results. NetLogo uses this library if it is available. So if you need strictly reproducible floating point math, you must run NetLogo in a Java VM that supports Java 1.3 or higher. NetLogo's About NetLogo dialog indicates whether this strict math mode is in effect.

- We have expended every effort to make NetLogo model runs fully reproducible, but of course this can never truly be an ironclad guarantee, due to the possibility of random hardware failure, and also due to the possibility of human error in the design of: your model, NetLogo, your Java VM, your hardware, and so on.

Downloading

The download form doesn't work for me. Can I have a direct link to the software?

Please write us at bugs@ccl.northwestern.edu and we'll either fix the problem with the form, or provide you with an alternate method of downloading the software.

Downloading NetLogo takes too long. Is it available any other way, such as on a CD?

At present, no. If this is a problem for you, contact us at feedback@ccl.northwestern.edu.

I downloaded and installed NetLogo but the Models Library has few or no models in it. How can I fix this?

So far, users reporting this problem all used the without VM download option for Windows. Uninstall NetLogo and try the with VM download instead.

Even if the with VM download fixes it for you, please contact us at bugs@ccl.northwestern.edu so we can find out more details about your setup. We'd like to fix this in a future version, but to troubleshoot it we need help from users.

Can I have multiple versions of NetLogo installed at the same time?

Yes. When you install NetLogo, the folder that is created contains has the version number in its name, so multiple versions can coexist.

On Windows systems, whichever version you installed last will be the version that opens when you double click a model file in Windows Explorer. On Mac OS 9, it's unpredictable what version opens. On Mac OS X, you can control it via Get Info in the Finder.

Usage

How do I change how many patches there are?

Select the Graphics Window by dragging a rectangle around it with the mouse. Click the Edit button in the Toolbar and type in new values for Screen Edge X and Screen Edge Y. (You can also right-click [Windows] or control-click [Mac] on the Graphics Window to edit it, or select it then double-click.)

How big can my model be? How many turtles, patches, procedures, buttons, and so on can my model contain?

The NetLogo engine has no fixed limits on size.

By default, though, on non-Macintosh operating systems NetLogo ships with a 512 megabyte ceiling on how much total RAM your model can use. But you can raise that limit by editing this section of the NetLogo.lax file in the NetLogo folder:

```
# LAX.NL.JAVA.OPTION.JAVA.HEAP.SIZE.MAX
# -----
# allow the heap to get huge
lax.nl.java.option.java.heap.size.max=536870912
```

If you are running NetLogo on Mac OS X in the Classic environment, then your model can't use more than 100MB of RAM. (If you are also running other applications in the Classic at the same time, then the ceiling may be lower.) This is due a limitation of the Classic environment itself. If you need to run very large models on Mac OS X, then you should use the native application, not the Classic application.

We have tested NetLogo with models that use hundreds of megabytes of RAM and they work fine. We haven't tested models that use gigabytes of RAM, though. It should work, but eventually perhaps you might hit some limits that are inherent in the underlying Java VM virtual machine and/or operating system (either built-in limits, or just bugs related to extremely high memory use).

Can I import a graphic into NetLogo?

At present, no. We plan to add this capability in a future version.

My model runs slowly. How can I speed it up?

Here's some ways to make it run faster without changing the code:

- If your model is using all available RAM on your computer, then installing more RAM should help. If your hard drive makes a lot of noise while your model is running, you probably need more RAM.
- If you are on Windows, make sure you are running with the IBM 1.1 VM. (You can check in the About NetLogo window.) This is the VM that is bundled with NetLogo if you choose the With VM download option. NetLogo's graphics are much faster with this VM than any other VM. If you are not using the IBM 1.1 VM, then the next tip will be especially helpful:
- Use the display switch in the graphics control strip, or the `no-display` primitive, to turn graphics off temporarily.

In many cases, though, if you want your model to run faster, you may need to make some changes to the code. Usually the most obvious opportunity for speedup is that you're doing too many computations that involve all the turtles or all the patches. Often this can be reduced by reworking the model so that it does less computation per time step. If you need help with this, if you contact us at feedback@ccl.northwestern.edu we may be able to help if you can send us your model or give us some idea of how it works. The members of the [NetLogo Users Group](#) may be able to help as well.

I want to try HubNet. Can I?

Currently, there are two implementations of [HubNet](#). One uses Texas Instruments calculators and requires Texas Instruments calculator network system, called [TI-Navigator](#). The other uses networked computers and does not require the TI-Navigator system.

TI recently announced that TI-Navigator is commercially available. However, HubNet does not work with this version of TI-Navigator; it might only work with a future released version of TI-Navigator.

The other implementation of HubNet is included in the NetLogo download and can be used without additional materials. Please refer to the [HubNet Guide](#) for more information on this version of HubNet.

Can I run a NetLogo model from the command line? Can I run it without a GUI?

At present there is no way to run without the GUI. We plan to add this in a future version. (On X11-based systems such as Linux, though, one possible solution to use X11s virtual frame buffer feature, which provides a virtual display for NetLogo to run on.)

Automating runs from the command line is possible, but currently not officially supported or documented. If you need this for your work, it can be done, but some light Java programming is required. We can supply sample Java code that you can alter for your needs. Anyone interested in doing this, write us at feedback@ccl.northwestern.edu.

Can I have more than one model open at a time?

One instance of NetLogo can only have one model open at a time. (We plan to change this in a future version.)

You can have multiple models open by opening multiple instances of NetLogo, though. On Windows and Linux, simply start the application again. On Macs, you'll need to duplicate the application in the Finder, then open the copy. (This will use only a very small amount of additional disk space, since most of NetLogo is actually in the NetLogo.jar file, which is stored outside the application.)

Can I copy or save a picture of the graphics window?

NetLogo itself does not have this capability presently, but you can use the screen-capture capabilities of your operating system, or a third-party screen-capture package, to do this.

Can I make a movie of my model?

NetLogo itself does not have this capability presently, but there are many third-party software packages that let you capture movies from any running application.

Programming

How is the NetLogo language different from the StarLogoT language? How do I convert my StarLogoT model to NetLogo?

We don't have a document that specifically summarizes the differences between these two programs. If you have built models in StarLogoT before, then we suggest reading the [Programming Guide](#) section of this manual to learn about NetLogo, particularly the sections on Ask and

Agentsets. Looking at some of the sample models and code examples in the Models Library may help as well.

If you need any help converting your StarLogo or StarLogoT model to NetLogo, please feel free to get in touch with us at feedback@ccl.northwestern.edu. The [NetLogo Users Group](#) is also a good resource for getting help from other users.

The NetLogo world is a torus, that is, the edges of the screen are connected to each other, so turtles and patches wrap around. Can I use a different world topology: bounded, infinite plane, sphere, etc.?

Torus is the only topology directly supported by NetLogo, but you can often simulate a different topology without too much extra effort.

If you want the world to be a bounded rectangle, you may need to add some code to your model to enforce this. Often a helpful technique is to turn the edge patches a different color, so turtles can easily detect when they hit the edge. Also, there are no-wrap versions of the distance and towards primitives; these should help.

If you want your turtles to move over an infinite plane, you can simulate this by having the turtles keep track of their position on the infinite plane, then hide the turtle when it goes out of bounds. The Random Walk 360 model in the Models Library shows you how to code this.

Simulating a spherical or other topology might be difficult; we haven't seen a model that does this. (If you have one, please send it in!)

Does NetLogo have a command like StarLogos grab command?

We don't have such a command, although we plan to add one or perhaps several! In the meantime, though, you can use the `without-interruption` primitive to arrange exclusive interaction between agents. For example:

```
turtles-own [mate]
to setup
  ask turtles [ set mate nobody ]
end
to find-mate ;; turtle procedure
  locals [candidate]
  without-interruption
  [ if mate = nobody
    [ set candidate random-one-of other-turtles-here
      with [mate = nobody]
      if candidate != nobody
        [ set mate candidate
          set mate-of candidate self ] ] ]
end
```

Using `without-interruption` ensures that while a turtle is choosing a mate, all other agents are frozen. This makes it impossible for two turtles to choose the same mate.

I tried to put `-at` after the name of a variable, for example `variable-at -1 0`, but NetLogo won't let me. Why not?

This syntax was supported by StarLogoT and some beta versions of NetLogo, but was removed from NetLogo 1.0. Instead, for a patch variable write e.g. `pcolor-of patch-at -1 0`, and for a turtle variable write e.g. `color-of one-of turtles-at -1 0`.

Im getting numbers like 0.1000000004 and 0.799999999999 instead of 0.1 and 0.8. Why?

That math operations sometimes produce slightly inaccurate results like these is a limitation of IEEE floating point arithmetic in general. (NetLogo uses Java to do math, and Java implements the IEEE standard.) But there are some cases where NetLogo should compensate for it and doesnt. We fix them as we find them (or you report them), but in general, you shouldnt expect floating point arithmetic to produce absolutely exact answers.

The precision primitive is handy for rounding off numbers for display purposes.

If you encounter a specific instance of this problem and are not sure whether its an issue with NetLogo or an issue with floating point arithmetic in general, run it by us and well take a look.

Is there some simple way to multiply one list by another list or number without writing a procedure?

At present, no. We plan to add this capability in a future version.

Can I read data from a file?

At present, this is only possible through the Import World mechanism, but you cant read files containing arbitrary data. We plan to add this capability in a future version of NetLogo.

In the meantime, you may find the following workarounds useful:

- 1) Copy and paste the data into the Procedures window, store it in a variable, and then work with it from there.
- 2) Use Import World to import the data. This will require modifying your data to be in the format that import-world expects. You can use Export World to produce an example of the format, and then use that as a template.

Can I write data to a file?

NetLogo currently has three mechanisms for this:

- Export World menu item (and `export-world` command). This records the current state of the NetLogo, including all agents and variables, but not including the contents of plots.
- Export Plot and Export All Plots menu items (and `export-output` and `export-all-plots` commands). These record the contents of plots. (Note that if you need to record data that changes over time, you may wish to create a plot to hold that data, not necessarily because you want to look at the plot itself, but purely because you want to export the plot later.)
- Export Output menu item (and `export-output` command). First use the `show`, `print`, and `type` commands to make any kind of output you want show up in the Command Center, then use this to save the output of the file. Note however that for Windows users, the output portion of the command center has a roughly 28K limit. (There may also be a [much higher]

limit on other operating systems depending on your Java VM.)

We plan to add additional file-writing capabilities to a future version of NetLogo.

How can I keep two turtles from occupying the same patch?

How to best accomplish this depends somewhat on the details of how your model works. The following techniques may be helpful:

- A turtle can test whether there are other turtles on the patch it is standing on with `any other-turtles-here`.
- A turtle can test whether there are other turtles on the patch it is facing with `any turtles-at dx dy`. This checks the patch that the turtle would land on if it executed `fd 1`. Note however that if the turtles heading isnt a multiple of 90, then `fd 1` will not necessarily take the turtle to a new patch it might only move from one corner of a patch to the opposite corner of the same patch.

Sometimes code that looks like it should prevent two turtles from ever being on the same patch can fail because of unexpected interactions between the turtles as they all execute the code in parallel. In these situations, appropriate use of `without-interruption` can be helpful.

How can I use different patch neighborhoods (circular, Von Neumann, Moore, etc.)?

The `in-radius` primitives lets you access circular neighborhoods of any radius.

The `neighbors` primitive gives you a Moore neighborhood of radius 1, and the `neighbors4` primitive gives you a Von Neumann neighborhood of radius 1.

If you want a Moore or Von Neumann neighborhood of a different radius, or a different kind of neighborhood altogether, you can define it yourself, using the `at-points` primitive and/or other techniques. If the neighborhoods do not change over time, then the most efficient way to use them is to compute the neighborhoods only once, ahead of time, and store them in agentsets. See this URL for a discussion and example code:

<http://groups.yahoo.com/group/netlogo-users/message/377>.

Can I connect turtles with lines, to indicate connections between them?

Eventually we plan to support this directly but in the meantime there are various workarounds you can use. See this URL for a discussion and example code:

<http://groups.yahoo.com/group/netlogo-users/message/324>.

How does NetLogo decide when to switch from agent to agent when running code?

Turtles are scheduled for execution in ascending order by ID number. Patches are scheduled for execution by row: left to right within each row, and starting with the top row. In a future version of NetLogo, we plan to add an option for randomized scheduling.

Once scheduled, an agents turn ends only once it performs an action that affects the state of the world, such as moving, or creating a turtle, or changing the value of an agent variable. (Setting a local variable doesnt count.)

To prolong an agents turn, use the `without-interruption` command.

In general, we suggest you write your NetLogo code so that it does not depend on a particular scheduling mechanism. We make no guarantees that the scheduling algorithm will remain the same in future versions.

Primitives Dictionary

Alphabetical: [A](#) [B](#) [C](#) [D](#) [E](#) [F](#) [G](#) [H](#) [I](#) [J](#) [K](#) [L](#) [M](#) [N](#) [O](#) [P](#) [R](#) [S](#) [T](#) [U](#) [V](#) [W](#) [X](#) [Y](#) [?](#)

Categories: [Turtle](#) [Patch](#) [Agentset](#) [Color](#) [Control/Logic](#) [Display](#) [HubNet](#) [I/O](#) [List](#) [String](#) [Math](#) [Plotting](#)

Special: [Variables](#) [Keywords](#) [Constants](#)

Categories of Primitives

This is an approximate grouping. Remember that a turtle-related primitive might still be called by patches or observers, and vice versa. To see which agent (turtles, patches, observer) can actually run each command, consult each individual entry in the dictionary.

Turtle-related

[back](#) ([bk](#)) [BREED-at](#) [BREED-here](#) [clear-turtles](#) ([ct](#)) [create-BREED](#) [create-custom-BREED](#) [create-custom-turtles](#) ([cct](#)) [create-turtles](#) ([crt](#)) [die](#) [distance](#) [distance-nowrap](#) [distancexy](#) [distancexy-nowrap](#) [downhill](#) [downhill4](#) [dx](#) [dy](#) [forward](#) ([fd](#)) [hatch](#) [hideturtle](#) ([ht](#)) [home](#) [inspect](#) [is-turtle?](#) [jump](#) [left](#) ([lt](#)) [myself](#) [no-label](#) [nobody](#) [-of](#) [other-turtles-here](#) [other-BREED-here](#) [patch-here](#) [pen-down](#) ([pd](#)) [pen-up](#) ([pu](#)) [right](#) ([rt](#)) [self](#) [set-default-shape](#) [setxy](#) [showturtle](#) ([st](#)) [sprout](#) [stamp](#) [towards](#) [towards-nowrap](#) [towardsxy](#) [towardsxy-nowrap](#) [turtle](#) [turtles](#) [turtles-at](#) [turtles-here](#) [uphill](#) [value-from](#)

Patch-related primitives

[clear-patches](#) ([cp](#)) [diffuse](#) [diffuse4](#) [distance](#) [distance-nowrap](#) [distancexy](#) [distancexy-nowrap](#) [inspect](#) [is-patch?](#) [myself](#) [neighbors](#) [neighbors4](#) [no-label](#) [nobody](#) [nsum](#) [nsum4](#) [-of](#) [patch](#) [patch-at](#) [patch-here](#) [patches](#) [self](#) [sprout](#) [value-from](#)

Agentset primitives

[any](#) [ask](#) [at-points](#) [BREED-at](#) [BREED-here](#) [count](#) [histogram](#) [in-radius](#) [is-agent?](#) [is-agentset?](#) [is-patch-agentset?](#) [is-turtle-agentset?](#) [max-one-of](#) [min-one-of](#) [neighbors](#) [neighbors4](#) [one-of](#) [other-turtles-here](#) [other-BREED-here](#) [patches](#) [random-n-of](#) [random-one-of](#) [turtles](#) [with](#) [turtles-at](#) [turtles-here](#) [values-from](#)

Color primitives

[extract-hsb](#) [extract-rgb](#) [hsb](#) [rgb](#) [scale-color](#) [shade-of?](#) [wrap-color](#)

Control flow and logic primitives

[and](#) [end](#) [foreach](#) [if](#) [ifelse](#) [locals](#) [loop](#) [map](#) [not](#) [or](#) [repeat](#) [report](#) [run](#) [runresult](#) [:](#) ([semicolon](#)) [stop](#) [startup](#) [to](#) [to-report](#) [wait](#) [while](#) [without-interruption](#) [xor](#)

Display primitives

[clear-all](#) ([ca](#)) [clear-graphics](#) ([cg](#)) [clear-patches](#) ([cp](#)) [clear-turtles](#) ([ct](#)) [display](#) [no-display](#) [no-label](#) [screen-edge-x](#) [screen-edge-y](#) [screen-size-x](#) [screen-size-y](#)

HubNet primitives

[hubnet-broadcast](#) [hubnet-enter-message?](#) [hubnet-exit-message?](#) [hubnet-fetch-message](#) [hubnet-message](#) [hubnet-message-source](#) [hubnet-message-tag](#) [hubnet-message-waiting?](#) [hubnet-reset](#) [hubnet-send](#) [hubnet-set-client-interface](#)

Input/output primitives

[clear-output](#) [export-output](#) [export-plot](#) [export-all-plots](#) [export-world](#) [get-date-and-time](#) [import-world](#) [mouse-down?](#) [mouse-xcor](#) [mouse-ycor](#) [print](#) [read-from-string](#) [reset-timer](#) [show](#) [timer](#) [type](#) [user-choice](#) [user-input](#) [user-message](#) [user-yes-or-no](#)

List primitives

[but-first](#) [but-last](#) [empty?](#) [filter](#) [first](#) [foreach](#) [fput](#) [is-list?](#) [item](#) [last](#) [length](#) [list](#) [lput](#) [map](#) [member?](#) [position](#) [reduce](#) [remove](#) [remove-duplicates](#) [replace-item](#) [reverse](#) [sentence](#) [sort](#) [sort-by](#) [values-from](#)

String primitives

[Operators](#) ([+](#), [<](#), [>](#), [=](#), [!=](#), [<=](#), [>=](#)) [but-first](#) [but-last](#) [empty?](#) [first](#) [is-string?](#) [item](#) [last](#) [length](#) [member?](#) [position](#) [remove](#) [read-from-string](#) [replace-item](#) [reverse](#) [substring](#) [word](#)

Mathematical primitives

[Arithmetic Operators](#) ([+](#), [*](#), [-](#), [/](#), [^](#), [<](#), [>](#), [=](#), [!=](#), [<=](#), [>=](#)) [abs](#) [acos](#) [asin](#) [atan](#) [ceiling](#) [cos](#) [e](#) [exp](#) [floor](#) [int](#) [ln](#) [log](#) [max](#) [mean](#) [median](#) [min](#) [mod](#) [pi](#) [precision](#) [random](#) [random-exponential](#) [random-normal](#) [random-poisson](#) [random-seed](#) [remainder](#) [round](#) [sin](#) [sqrt](#) [standard-deviation](#) [sum](#) [tan](#) [variance](#)

Plotting primitives

[autoplot?](#) [auto-plot-off](#) [auto-plot-on](#) [clear-all-plots](#) [clear-plot](#) [create-temporary-plot-pen](#) [export-plot](#) [export-all-plots](#) [histogram](#) [histogram-list](#) [plot](#) [plot-name](#) [plot-pen-down](#) ([ppd](#)) [plot-pen-reset](#) [plot-pen-up](#) ([ppu](#)) [plot-x-max](#) [plot-x-min](#) [plot-y-max](#) [plot-y-min](#) [plotxy](#) [ppd](#) [ppu](#) [set-current-plot](#) [set-current-plot-pen](#) [set-histogram-num-bars](#) [set-plot-pen-color](#) [set-plot-pen-interval](#) [set-plot-pen-mode](#) [set-plot-x-range](#) [set-plot-y-range](#)

Built-In Variables

Turtles

[breed](#) [color](#) [heading](#) [hidden?](#) [label](#) [label-color](#) [pen-down?](#) [shape](#) [size](#) [who](#) [xcor](#) [ycor](#)

Patches

[pcolor](#) [plabel](#) [plabel-color](#) [pxcor](#) [pycor](#)

Other

[?](#)

Keywords

[breeds](#) [end](#) [globals](#) [locals](#) [patches-own](#) [to](#) [to-report](#) [turtles-own](#)

Constants

Mathematical Constants

e = 2.718281828459045

pi = 3.141592653589793

Boolean Constants

false

true

Color Constants

The allowable range of values for colors is 0 up to but not including 140. Each color ranges from black to white over a scale of ten. Thus the color red goes from black (10) to dark red (11) to red (15) to light red (19) to white (19.9). The scale is discontinuous; 19.9 is white, but 20.0 is black.

The available color names are listed below. (See also the [rgb](#) and [hsb](#) primitives.)

black = 0

gray = 5

white = 9.999

red = 15

orange = 25

brown = 35

yellow = 45

green = 55

lime = 65

turquoise = 75

cyan = 85

```
sky = 95
blue = 105
violet = 115
magenta = 125
pink = 135
```

A

abs

abs number

Reports the absolute value of *number*.

```
show abs -7
=> 7
show abs 5
=> 5
```

acos

acos number

Reports the arc cosine (inverse cosine) of the given number. The input must be in the range -1.0 to 1.0 . The result is in degrees, and lies in the range 0.0 to 180.0 .

and

condition1 and condition2

Reports true if both *condition1* and *condition2* are true.

Note that if *condition1* is false, then *condition2* will not be run (since it can't affect the result).

```
if (pxcor > 0) and (pycor > 0)
  [ set pcolor blue ] ;; the upper-right quadrant of
                      ;; patches turn blue
```

any

any agentset

Reports true if the given agentset is non-empty, false otherwise.

Equivalent to `count agentset > 0`, but arguably more readable.

```
if any turtles with [color = red]
  [ show "at least one turtle is red!" ]
```

Arithmetic Operators (+, *, -, /, ^, <, >, =, !=, <=, >=)

All of these operators take two inputs, and all act as infix operators (going between the two inputs, as in standard mathematical use). NetLogo correctly supports order of operations for infix operators.

The operators work as follows: + is addition, * is multiplication, - is subtraction, / is division, ^ is exponentiation, < is less than, > is greater than, = is equal to, != is not equal to, <= is less than or equal, >= is greater than or equal.

All of the comparison operators also work on strings, and the addition operator (+) also functions as a string concatenation operator (see example below).

If you are not sure how NetLogo will interpret your code, you should insert parentheses.

```
show 5 * 6 + 6 / 3
=> 32
show 5 * (6 + 6) / 3
=> 20
show "tur" + "tle"
=> turtle
```

asin

asin number

Reports the arc sine (inverse sine) of the given number. The input must be in the range -1.0 to 1.0. The result is in degrees, and lies in the range -90.0 to 90.0.

ask

ask agentset [commands]

ask agent [commands]

Takes a list of commands that will be run by the specified agent or agentset.

```
ask turtles [ fd 1 ]
  ;; all turtles move forward one step
ask patches [ set pcolor red ]
  ;; all patches turn red
ask turtle 4 [ rt 90 ]
  ;; only the turtle with id 4 turns right
```

at-points

agentset at-points [[x1 y1] [x2 y2] ...]

Reports a subset of the given agentset that includes only the agents on the patches the given distances away from the calling agent. The distances are specified as a list of two-item lists, where the two items are the x and y offsets.

If the caller is the observer, then the points are measured relative to the origin, in other words, the points are taken as absolute patch coordinates.

If the caller is a turtle, the points are measured relative to the turtles exact location, and not from the center of the patch under the turtle.

```
ask turtles at-points [[2 4] [1 2] [10 15]]
[ fd 1 ] ;; only the turtles on the patches at the
         ;; distances (2,4), (1,2) and (10,15),
         ;; relative to the caller, move
```

atan

atan x y

Reports the arc tangent, in degrees (from 0 to 360), of x divided by y.

When y is 0: if x is positive, it reports 90; if x is zero, it reports 0; if x is negative, it reports 270.

Note that this version of atan is designed to conform to the geometry of the NetLogo world, where a heading of 0 is straight up, 90 is to the right, and so on clockwise around the circle. (Normally in geometry an angle of 0 is right, 90 is up, and so on, counterclockwise around the circle, and atan would be defined accordingly.)

```
show atan 1 -1
=> 135.0
show atan -1 1
=> 315.0
```

autoplot?

autoplot?

Reports true if auto-plotting is on for the current plot, false otherwise.

auto-plot-off

auto-plot-on

auto-plot-off

auto-plot-on

This pair of commands is used to control the NetLogo feature of auto-plotting in the current plot. Auto-plotting will automatically update the x and y axes of the plot whenever the current pen exceeds these boundaries. It is useful when wanting to display all plotted values in the current plot, regardless of the current plot ranges.

B

back

bk

back number

The turtle moves backward by *number* steps. (If *number* is negative, the turtle moves forward.)

Turtles using this primitive can move a maximum of one unit per time increment. So `bk 0.5` and `bk 1` both take one unit of time, but `bk 3` takes three.

See also [forward](#), [jump](#).

breed**breed**

This is a built-in turtle variable. It holds the agentset of all turtles of the same breed as this turtle. (For turtles that do not have any particular breed, this is the [turtles](#) agentset of all turtles.) You can set this variable to change a turtles breed.

See also [breeds](#).

Example:

```
breeds [cats dogs]
;; turtle code:
if breed = cats [ show "meow!" ]
set breed dogs
show "woof!"
```

breeds**breeds [*breed1 breed2 ...*]**

This keyword, like the globals, turtles-own, and patches-own keywords, can only be used at the beginning of a program, before any function definitions. It defines breeds and their associated agentsets.

Any turtle of the given breed:

- is part of the agentset named by the breed name
- has its breed built-in variable set to that agentset

Most often, the agentset is used in conjunction with `ask` to give commands to only the turtles of a particular breed.

```
breeds [ mice frogs ]
to setup
  ca
  create-mice 50
  ask mice [ set color white ]
  create-frogs 50
  ask frogs [ set color green ]
```

```

show breed-of one-of mice    ;; prints mice
show breed-of one-of frogs  ;; prints frogs
end

```

See also [globals](#), [patches-own](#), [turtles-own](#), [<BREED>-own](#), [create-<BREED>](#), [create-custom-<BREED>](#), [<BREED>-at](#), [<BREED>-here](#).

but-first

bf

but-last

bl

but-first *list*

but-first *string*

but-last *list*

but-last *string*

When used on a list, but-first reports all of the list items of *list* except the first, and but-last reports all of the list items of *list* except the last.

On strings, but-first and but-last report a shorter string omitting the first or last character of the original string.

```

;; mylist is [2 4 6 5 8 12]
set mylist but-first mylist
;; mylist is now [4 6 5 8 12]
set mylist but-last mylist
;; mylist is now [4 6 8]
show but-first "string"
;; prints "tring"
show but-last "string"
;; prints "strin"

```

C

ceiling

ceiling *number*

Reports the smallest integer greater than or equal to *number*.

```

show ceiling 4.5
=> 5
show ceiling -4.5
=> -4

```

clear-all

ca

clear-all

Kills all turtles, resets all global variables to zero, and calls `clear-patches` and `clear-all-plots`.

clear-all-plots**clear-all-plots**

Clears every plot in the model. See [clear-plot](#) for more information.

clear-graphics**cg****clear-graphics**

Kills all turtles and clears all patches. Combines the effect of [clear-turtles](#) and [clear-patches](#).

clear-output**cc****clear-output**

Clears all text from the output portion of the Command Center.

clear-patches**cp****clear-patches**

Clears the patches by resetting all patch variables to their default initial values, including setting their color to black.

clear-plot**clear-plot**

In the current plot only, resets all plot pens, deletes all temporary plot pens, resets the plot to its default values (for x range, y range, etc.), and resets all permanent plot pens to their default values. The default values for the plot and for the permanent plot pens are set in the plot Edit dialog, which is displayed when you edit the plot. If there are no plot pens after deleting all temporary pens, that is to say if there are no permanent plot pens, a default plot pen will be created with the following initial settings:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Name: default
- Interval: 1.0

See also [clear-all-plots](#).

clear-turtles ct

clear-turtles



Kills all turtles.

See also [die](#).

color

color



This is a built-in turtle variable. It holds the color of the turtle. You can set this variable to make the turtle change color.

See also [pcolor](#).

COS

cos *number*

Reports the cosine of the given angle. Assumes the angle is given in degrees.

```
show cos 180
=> -1.0
```

count

count *agentset*

Reports the number of agents in the given agentset.

```
show count turtles
;; prints the total number of turtles
show count patches with [pcolor = red]
;; prints the total number of red patches
```

create-turtles**crt****create-<BREED>****create-turtles *number*****create-<BREED> *number***

Creates *number* new turtles . New turtles start at position (0, 0), are created with the 14 primary colors, and have headings from 0 to 360, evenly spaced.

```
crt 100
ask turtles [ fd 10 ] ;; makes an evenly spaced circle
```

If the create-<BREED> form is used, the new turtles are created as members of the given breed.

create-custom-turtles**cct****create-custom-<BREED>****cct-<BREED>****create-custom-turtles *number* [*commands*]****create-custom-<BREED> *number* [*commands*]**

Creates *number* new turtles (of the given breed, if specified). New turtles start at position (0, 0). New turtles are created with the 14 primary colors and have headings from 0 to 360, evenly spaced.

The new turtles immediately run *commands*. This is useful for giving the new turtles a different color, heading, or whatever.

```
breeds [canaries snakes]
to setup
  ca
  create-custom-canaries 50
  [ set color yellow ]
  create-custom-snakes 50
  [ set color green ]
end
```

Note: While the commands are running, no other agents are allowed to run any code (as with the without-interruption command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no screen updates take place until the commands are done. This ensures that the new turtles are never drawn on-screen until they are fully initialized.

create-temporary-plot-pen

create-temporary-plot-pen *string*

A new temporary plot pen with the given name is created in the current plot and set to be the current pen.

Few models will want to use this primitive, because all temporary pens disappear when `clear-plot` or `clear-all-plots` are called. The normal way to make a pen is to make a permanent pen in the plots Edit dialog.

If a temporary pen with that name already exists in the current plot, no new pen is created, and the existing pen is set to be the current pen. If a permanent pen with that name already exists in the current plot, a runtime error is thrown.

The new temporary plot pen has the following initial settings:

- Pen: down
- Color: black
- Mode: 0 (line mode)
- Interval: 1.0

Note that the old primitive `set-plot-pen`, which was used in earlier versions of NetLogo, is now just an alternate name for this primitive. `set-plot-pen` is available only for reasons of compatibility; new models should not use `set-plot-pen`.

See: [clear-plot](#), [clear-all-plots](#), and [set-current-plot-pen](#).

D

die**die**

The turtle dies.

```
if xcor > 20 [ die ]
;; all turtles with xcor greater than 20 die
```

See also: [ct](#)

diffuse**diffuse *patch-variable number***

Tells each patch to share (*number* * 100) percent of the value of *patch-variable* with its eight neighboring patches. *number* should be between 0 and 1.

Note that this is an observer command only, even though you might expect it to be a patch command. (The reason is that it acts on all the patches at once patch commands act on individual patches.)

```
diffuse chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 8 patches. Thus,
;; each patch gets 1/8 of 50% of the chemical
;; from each neighboring patch.)
```

diffuse4

diffuse4 *patch-variable number*



Like `diffuse`, but only diffuses to the four neighboring patches (to the north, south, east, and west), not to the diagonal neighbors.

```
diffuse4 chemical 0.5
;; each patch diffuses 50% of its variable
;; chemical to its neighboring 4 patches. Thus,
;; each patch gets 1/4 of 50% of the chemical
;; from each neighboring patch.)
```

display

display



Resumes updating of the graphics window (which may have been turned off using the `no-display` command). Any changes in the state of the graphics window that were made while `no-display` was in effect will immediately be drawn on-screen all at once.

Note that `display` and `no-display` operate independently of the switch in the graphics window control strip that freezes the display.

See also [no-display](#).

distance

distance *agent*



Reports the distance from this agent to the given turtle or patch.

The distance to or a from a patch is measured from the center of the patch.

Unlike `distance-nowrap`, turtles and patches use the wrapped distance (around the edges of the screen) if that distance is shorter than the on-screen distance.

distance-nowrap

distance-nowrap *agent*



Reports the distance from this agent to the given turtle or patch.

The distance to or a from a patch is measured from the center of the patch.

Unlike distance, this always reports the on-screen distance, never a distance that would require wrapping around the edges of the screen.

distancexy

distancexy *xcor ycor*



Reports the distance from this agent to the point (*xcor*, *ycor*).

The distance from a patch is measured from the center of the patch.

Unlike distance-nowrap, the wrapped distance (around the edges of the screen) is used if that distance is shorter than the on-screen distance.

```
if (distancexy 0 0) > 10
  [ set color green ]
;; all turtles more than 10 units from
;; the center of the screen turn green.
```

distancexy-nowrap

distancexy-nowrap *xcor ycor*



Reports the distance from this agent to the point (*xcor*, *ycor*).

The distance from a patch is measured from the center of the patch.

Unlike distancexy, this always reports the on-screen distance, never a distance that would require wrapping around the edges of the screen.

downhill

downhill *patch-variable*



Reports the turtle heading (between 0 and 359 degrees) in the direction of the minimum value of the variable *patch-variable*, of the patches in a one-patch radius of the turtle. (This could be as many as eight or as few as five patches, depending on the position of the turtle within its patch.)

If there are multiple patches that have the same smallest value, a random one of those patches will be selected.

If the patch is located directly to the north, south, east, or west of the patch that the turtle is currently on, a multiple of 90 degrees is reported. However, if the patch is located to the northeast, northwest, southeast, or southwest of the patch that the turtle is currently on, the direction the turtle would need to reach the nearest corner of that patch is reported.

See also [downhill4](#), [uphill](#), [uphill4](#).

downhill4

downhill4 *patch-variable*



Reports the turtle heading (between 0 and 359 degrees) as a multiple of 90 degrees in the direction of the minimum value of the variable *patch-variable*, of the four patches to the north, south, east, and west of the turtle. If there are multiple patches that have the same least value, a random patch from those patches will be selected.

See also [downhill](#), [uphill](#), [uphill4](#).

dx

dy

dx

dy



Reports the x-increment or y-increment if the turtle were to take one step forward in its current heading.

These primitives are useful for testing the patch ahead of the turtle before moving.

```
if (food-of patch-at dx dy) > 0
  [ fd 1
    eat ]
```

E

empty?

empty? *list*

empty? *string*

Reports true if the given list or string is empty, false otherwise.

Note: the empty list is written `[]`. The empty string is written `" "`.

end**end**

Used to conclude a procedure. See [to](#) and [to-report](#).

every**every *number* [*commands*]**

Runs the given commands at most every *number* seconds.

By itself, `every` doesn't make commands run over and over again. You need to use `every` inside a loop, or inside a `forever` button, if you want the commands run over and over again. `every` only limits how often the commands run.

More technically, its exact behavior is as follows. When an agent reaches an `every`, it checks a timer to see if the given amount of time has passed since the last time the same agent ran the commands in the `every` in the same context. If so, it runs the commands; otherwise they are skipped and execution continues.

Here, in the same context means during the same `ask` (or button press or command typed in the Command Center). So it doesn't make sense to write `ask turtles [every 0.5 [...]]`, because when the `ask` finishes the turtles will all discard their timers for the `every`. The correct usage is shown below.

```
every 0.5 [ ask turtles [ fd 1 ] ]
;; twice a second the turtles will move forward 1
every 2 [ set index index + 1 ]
;; every 2 seconds index is incremented
```

See also [wait](#).

exp**exp *number***

Reports the value of e raised to the *number* power.

Note: This is the same as $e \wedge \textit{number}$.

export-output**export-plot****export-all-plots****export-world**

export-output *filename*
export-plot *plotname filename*
export-all-plots *filename*
export-world *filename*

export-output writes the contents of the output portion of the Command Center to an external file given by the string *filename*.

export-plot writes the x and y values of all points plotted by all the plot pens in the plot given by the string *plotname* to an external file given by the string *filename*. If a pen is in bar mode (mode 0) and the y value of the point plotted is greater than 0, the upper-left corner point of the bar will be exported. If the y value is less than 0, then the lower-left corner point of the bar will be exported.

export-all-plots writes every plot in the current model to an external file given by the string *filename*. Each plot is identical in format to the output of export-plot.

export-world writes the values of all variables, both built-in and user-defined, including all observer, turtle, and patch variables, to an external file given by the string *filename*. (The result file can be read back into NetLogo with the [import-world](#) primitive.)

export-plot, export-all-plots and export-world save files in in plain-text, comma-separated values (.csv) format. CSV files can be read by most popular spreadsheet and database programs as well as any text editor.

If the file already exists, it is overwritten.

If you wish to export to a file in a location other than the NetLogo folder, you should include the full path to the file you wish to export. (Use the forward-slash / as the folder separator.)

Note that the functionality of these primitives is also available directly from NetLogos File menu.

```
export-world "fire.csv"
;; exports the state of the model to the file fire.csv
;; located in the NetLogo folder
export-plot "Temperature" "c:/My Documents/plot.csv"
;; exports the plot named
;; "Temperature" to the file plot.csv located in
;; the C:\My Documents folder
export-all-plots "c:/My Documents/plots.csv"
;; exports all plots to the file plots.csv
;; located in the C:\My Documents folder
```

extract-hsb

extract-hsb *color*

Reports a list of three values in the range 0.0 to 1.0 representing the hue, saturation and brightness, respectively, of the given NetLogo *color* in the range 0 to 140.

```
show extract-hsb red
=> [0.0 1.0 1.0]
show extract-hsb cyan
=> [0.5 1.0 1.0]
```

See also [hsb](#), [rgb](#), [extract-rgb](#).

extract-rgb

extract-rgb *color*

Reports a list of three values in the range 0.0 to 1.0 representing the levels of red, green, and blue, respectively, of the given NetLogo *color* in the range 0 to 140.

```
show extract-rgb red
=> [1.0 0.0 0.0]
show extract-rgb cyan
=> [0.0 1.0 1.0]
```

See also [rgb](#), [hsb](#), [extract-hsb](#).

F

filter

filter [*reporter*] *list*

Reports a list containing only those items of *list* for which the boolean *reporter* is true in other words, the items satisfying the given condition.

In *reporter*, use [?](#) to refer to the current item of *list*.

```
show filter [? < 3] [1 3 2]
=> [1 2]
show filter [first ? != "t"] ["hi" "there" "everyone"]
=> ["hi" "everyone"]
```

See also [map](#), [reduce](#), [?](#).

first

first *list*

first *string*

On a list, reports the first (0th) item in the list.

On a string, reports a one-character string containing only the first character of the original string.

floor

floor *number*

Reports the largest integer less than or equal to *number*.

```
show floor 4.5
=> 4
```

```
show floor -4.5
=> -5
```

foreach

foreach *list* [*commands*]
(foreach *list1* ... *listn* [*commands*])

With a single list, runs *commands* for each item of *list*. In *commands*, use [?](#) to refer to the current item of *list*.

```
foreach [1.1 2.2 2.6] [ show ? + " -> " + round ? ]
=> 1.1 -> 1
=> 2.2 -> 2
=> 2.6 -> 3
```

With multiple lists, runs *commands* for each group of items from each list. So, they are run once for the first items, once for the second items, and so on. All the lists must be the same length. In *commands*, use [?1](#) through [?n](#) to refer to the current item of each list.

Some examples make this clearer:

```
(foreach [1 2 3] [2 4 6] [ show "the sum is: " + (?1 + ?2) ])
=> the sum is: 3
=> the sum is: 6
=> the sum is: 9
(foreach list (turtle 1) (turtle 2) [3 4] [ ask ?1 [ fd ?2 ] ])
;; turtle 1 moves forward 3 patches
;; turtle 2 moves forward 4 patches
```

See also [map](#), [?](#).

forward

fd

forward *number*



The turtle moves forward by *number* steps. (If *number* is negative, the turtle moves backward.)

Turtles using this primitive can move a maximum of one unit per time increment. So `fd 0.5` and `fd 1` both take one unit of time, but `fd 3` takes three.

See also [jump](#).

fput

fput *item list*

Adds *item* to the beginning of a list and reports the new list.

```
;; suppose mylist is [5 7 10]
```

```
set mylist fput 2 mylist
;; mylist is now [2 5 7 10]
```

G

get-date-and-time

get-date-and-time

Reports a string containing the current date and time. The format is shown below. All fields are fixed width, so they are always at the same locations in the string. The potential resolution of the clock is milliseconds. (Whether you get resolution that high in practice may vary from system to system, depending on the capabilities of the underlying Java Virtual Machine.)

```
show get-date-and-time
=> 01:19:36.685 PM 19-Sep-2002
```

globals

globals [*var1 var2 ...*]

This keyword, like the breeds, *<BREED>-own*, *patches-own*, and *turtles-own* keywords, can only be used at the beginning of a program, before any function definitions. It defines new global variables. Global variables are global because they are accessible by all agents and can be used anywhere in a model.

Most often, *globals* is used to define variables or constants that need to be used in many parts of the program.

H

hatch

hatch *number* [*commands*]



Each turtle creates *number* new turtles, each identical to itself, and tells the new turtles to run *commands*. This is useful for giving the new turtles different colors, headings, breeds, or whatever.

Note: While the commands are running, no other agents are allowed to run any code (as with the *without-interruption* command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no screen updates take place until the commands are done. This ensures that the new turtles are never drawn on-screen in an only partly initialized state.

```
hatch 1 [ lt 45 fd 1 ]
;; each turtle creates one new turtle,
;; and the child turns and moves away
```

heading

heading



This is a built-in turtle variable. It indicates the direction the turtle is facing. This is a number greater than or equal to 0 and less than 360. 0 is north, 90 is east, and so on. You can set this variable to make a turtle turn.

See also [right](#), [left](#), [dx](#), [dy](#).

Example:

```
set heading 45      ;; turtle is now facing northeast
set heading heading + 10 ;; same effect as "rt 10"
```

hidden?

hidden?



This is a built-in turtle variable. It holds a boolean (true or false) value indicating whether the turtle is currently hidden (i.e., invisible). You can set this variable to make a turtle disappear or reappear.

See also [hideturtle](#), [showturtle](#).

Example:

```
set hidden? not hidden?
;; if turtle was showing, it hides, and if it was hiding,
;; it reappears
```

hideturtle

ht

hideturtle



The turtle makes itself invisible.

Note: This command is equivalent to setting the turtle variable `hidden?` to true.

See also [showturtle](#).

histogram

histogram *agentset* [*reporter*]

Removes points drawn by the current plot pen, then draws a histogram of the values reported when all agents in the agentset run the given reporter. (It should report a numeric value. Any non-numeric

values reported are ignored.)

The histogram is drawn on the current plot using the current plot pen and pen color. Use `set-plot-x-range` to control the range of values to be histogrammed, and set the pen interval (either directly with `set-plot-pen-interval`, or indirectly via `set-histogram-num-bars`) to control how many bars that range is split up into.

Be sure that if you want the histogram drawn with bars that the current pen is in bar mode (mode 1).

```

histogram turtles [color]
;; draws a histogram showing how many turtles there are
;; of each color

```

Note: using this primitive amounts to the same thing as writing: `histogram-list values-from agentset [reporter]`, but is more efficient.

histogram-list

histogram-list *list*

Removes points drawn by the current plot pen, then draws a histogram of the values in the given list.

See [histogram](#), above, for more information.

home

home



Returns turtle to the origin. Equivalent to `setxy 0 0`.

hsb

hsb *hue saturation brightness*

Reports a number in the range 0 to 140, not including 140 itself, that represents the given color, specified in the HSB spectrum, in NetLogos color space.

All three values should be in the range 0.0 to 1.0.

The color reported may be only an approximation, since the NetLogo color space does not include all possible colors. (It contains only certain discrete hues, and for each hue, either saturation or brightness may vary, but not both – at least one of the two is always 1.0.)

```

show hsb 0 0 0
=> 0.0 ;; (black)
show hsb 0.5 1.0 1.0
=> 85.0 ;; (cyan)

```

See also [extract-hsb](#), [rgb](#), [extract-rgb](#).

hubnet-broadcast

hubnet-broadcast *tag-name value*

This broadcasts *value* from NetLogo to the variable, in the case of Calculator HubNet, or interface element, in the case of Computer HubNet, with the name *tag-name* to the clients.

See the [HubNet Authoring Guide](#) for details and instructions.

hubnet-enter-message?

hubnet-enter-message?

Reports true if a new computer client just entered the simulation. Reports false otherwise. [hubnet-message-source](#) will contain the user id of the client that just logged on.

See the [HubNet Authoring Guide](#) for details and instructions.

hubnet-exit-message?

hubnet-exit-message?

Reports true if a new computer client just exited the simulation. Reports false otherwise. [hubnet-message-source](#) will contain the user id of the client that just logged off.

See the [HubNet Authoring Guide](#) for details and instructions.

hubnet-fetch-message

hubnet-fetch-message

If there is any new data sent by the clients, this retrieves the next piece of data, so that it can be accessed by [hubnet-message](#).

See the [HubNet Authoring Guide](#) for details.

hubnet-message

hubnet-message

Reports the message retrieved by [hubnet-fetch-message](#).

See the [HubNet Authoring Guide](#) for details.

hubnet-message-source

hubnet-message-source

Reports the id of the client that sent the message retrieved by [hubnet-fetch-message](#).

See the [HubNet Authoring Guide](#) for details.

hubnet-message-tag

hubnet-message-tag

Reports the tag that is associated with the data that was retrieved by [hubnet-fetch-message](#). For Calculator HubNet, this will report one of the variable names set with the [hubnet-set-client-interface](#) primitive. For Computer HubNet, this will report one of the Display Names of the interface elements in the client interface.

See the [HubNet Authoring Guide](#) for details.

hubnet-message-waiting?

hubnet-message-waiting?

This looks for a new message sent by the clients. It reports true if there is one, and false if there is not.

See the [HubNet Authoring Guide](#) for details.

hubnet-reset

hubnet-reset

Logs you into the HubNet system. You must be logged in to use any of the other hubnet primitives with the exception of [hubnet-set-client-interface](#).

See the [HubNet Authoring Guide](#) for details.

hubnet-send

hubnet-send *string tag-name value*

hubnet-send *list-of-strings tag-name value*

For Calculator HubNet, this primitive acts in exactly the same manner as [hubnet-broadcast](#). (We plan to change this in a future version of NetLogo.)

For Computer HubNet, it acts as follows:

For a *string*, this sends *value* from NetLogo to the tag *tag-name* on the client that has *string* for its user id.

For a *list-of-strings*, this sends *value* from NetLogo to the tag *tag-name* on all the clients that have user ids that are in the *list-of-strings*.

See the [HubNet Authoring Guide](#) for details.

hubnet-set-client-interface

hubnet-set-client-interface *client-type client-info*

If *client-type* is TI-83+, *client-info* is a list containing two items. The first item is a string containing the name of the activity to enable on the TI Navigator web site. hubnet-set-client-interface TI-83+ notifies the user to enable this activity. The second item is a list of the tags for which to check. The tag list sets which variables NetLogo expects from the calculators. NetLogo will only check for these variables and will ignore all others.

If *client-type* is COMPUTER, *client-info* is a list containing a string with the file name and path (relative to NetLogo) to the file which will serve as the clients interface. This interface will be sent to any clients that log in.

```
hubnet-set-client-interface "TI-83+" [ "AAA - Gridlock 1.3" ["L1" "LOCS"] ]
;; notifies the user to enable the activity AAA - Gridlock 1.3 and looks for the
;; calculator lists L1 and LOCS on the Navigator server
```

```
hubnet-set-client-interface
  "COMPUTER"
  [ "models/HubNet Computer Activities/clients/Disease client.nlogo" ]
;; when clients log in, they will will get the interface described in the file
;; models/HubNet Computer Activities/clients/Disease client.nlogo
```

Future versions of HubNet may support other client types, and/or change the meaning of the second input to this command.

See the [HubNet Authoring Guide](#) for details.

I

if

if *condition* [*commands*]

Reporter must report a boolean (true or false) value.

If *condition* reports true, runs *commands*.

The reporter may report a different value for different agents, so some agents may run *commands* and others dont.

```
if xcor > 0[ set color blue ]
```

```
;; turtles on the right half of the screen
;; turn blue
```

ifelse

ifelse *reporter* [*commands1*] [*commands2*]

Reporter must report a boolean (true or false) value.

If *reporter* reports true, runs *commands1*.

If *reporter* reports false, runs *commands2*.

The reporter may report a different value for different agents, so some agents may run *commands1* while others run *commands2*.

```
ask patches
  [ ifelse pxcor > 0
    [ set pcolor blue ]
    [ set pcolor red ] ]
;; the left half of the screen turns red and
;; the right half turns blue
```

import-world

import-world *filename*

Reads the values of all variables for a model, both built-in and user-defined, including all observer, turtle, and patch variables, from an external file named by the given string. The file should be in the format used by the [export-world](#) primitive.

Note that the functionality of this primitive is also directly available from NetLogos File menu.

When using import-world, to avoid errors, perform these steps in the following order:

1. Open the model from which you created the export file.
2. Press the Setup button, to get the model in a state from which it can be run.
3. Import the file.
4. If you want, press Go button to continue running the model from the point where it left off.

If you wish to import a file from a location other than the NetLogo folder, you may include the full path to the file you wish to import. See [export-world](#) for an example.

in-radius

agentset in-radius *number*



Reports an agentset that includes only those agents from the original agentset whose distance from the caller is less than or equal to *number*.

The distance to or a from a patch is measured from the center of the patch.

```
ask turtles
  [ ask patches in-radius 3
    [ set pcolor red ] ]
;; each turtle makes a red "splotch" around itself
```

inspect

inspect *agent*

Opens an agent monitor for the given agent (turtle or patch).

```
inspect patch 2 4
;; an agent monitor opens for that patch
inspect random-one-of sheep
;; an agent monitor opens for a random turtle from
;; the "sheep" breed
```

int

int *number*

Reports the integer part of number any fractional part is discarded.

```
show int 4.7
=> 4
show int -3.5
=> -3
```

is-agent?

is-agentset?

is-boolean?

is-list?

is-number?

is-patch?

is-patch-agentset?

is-string?

is-turtle?

is-turtle-agentset?

is-agent? *value*

is-agentset? *value*

is-boolean? *value*

is-list? *value*

is-number? *value*

is-patch? *value*

is-patch-agentset? *value*

is-string? *value*

is-turtle? *value*

is-turtle-agentset? *value*

Reports true if *value* is of the given type, false otherwise.

item**item *index list*****item *index string***

On lists, reports the value of the item in the given list with the given index.

On strings, reports the character in the given string at the given index.

Note that the indices begin from 0, not 1. (The first item is item 0, the second item is item 1, and so on.)

```
;; suppose mylist is [2 4 6 8 10]
show item 2 mylist
=> 6
show item 3 "my-shoe"
=> "s"
```

J**jump****jump *number***

Turtles move forward by *number* units all at once, without the amount of time passing depending on the distance.

This command is useful for synchronizing turtle movements. The command forward 15 takes 15 times longer to run than forward 1, but jump 15 runs in the same amount of time as forward 1.

Note: When turtles jump, they do not step on any of the patches along their path.

L**label****label**

This is a built-in turtle variable. It may hold a value of any type. The turtle appears in the graphics window with the given value attached to it as text. You can set this variable to add, change, or remove a turtles label.

See also [no-label](#), [label-color](#), [plabel](#), [plabel-color](#).

Example:

```
ask turtles [ set label who ]
;; all the turtles now are labeled with their
;; id numbers
ask turtles [ set label no-label ]
;; all turtles now are not labeled
```

label-color

label-color



This is a built-in turtle variable. It holds a number greater than or equal to 0 and less than 140. This number determines what color the turtles label appears in (if it has a label). You can set this variable to change the color of a turtles label.

See also [no-label](#), [label](#), [p-label](#), [p-label-color](#).

Example:

```
ask turtles [ set label-color red ]
;; all the turtles now have red labels
```

last

last *list*

last *string*

On a list, reports the last item in the list.

On a string, reports a one-character string containing only the last character of the original string.

left

It

left *number*



The turtle turns left by *number* degrees. (If *number* is negative, it turns right.)

length

length *list*

length *string*

Reports the number of items in the given list, or the number of characters in the given string.

list

list *value1 value2*
(list *value1 ... valuen*)

Reports a list containing the given items. The items can be of any type, produced by any kind of reporter.

```
show list (random 10) (random 10)
=> [4 9] ;; or similar list
show (list (random 10) 1 2 3 (random 10))
=> [4 1 2 3 9] ;; or similar list
```

ln

ln *number*

Reports the natural logarithm of *number*, that is, the logarithm to the base e (2.71828...).

See also [e](#), [log](#).

locals

locals [*Vax1 var2 ...*]

Locals is a keyword used to declare local variables in a procedure, that is, variables that are usable only within that procedure. It must appear at the beginning of the procedure, before any commands.

A local variable may not have the same name as an existing observer, turtle, or patch variable.

See [to](#) and [to-report](#).

```
to hunt ;; turtle procedure
  locals [prey]
  set prey random-one-of other-turtles-here
  if prey != nobody
    [ eat prey ]
end
```

log

log *number base*

Reports the logarithm of *number* in base *base*.

```
show log 64 2
=> 6
```

See also [ln](#).

loop

loop [*commands*]

Runs the list of commands forever, or until the current procedure exits through use of the [stop](#) command or the [report](#) command.

Note: In most circumstances, you should use a forever button in order to repeat something forever. The advantage of using a forever button is that the user can click the button to stop the loop.

lput

lput *value list*

Adds *value* to the end of a list and reports the new list.

```
;; suppose mylist is [2 7 10 "Bob"]
set mylist lput 42 mylist
;; mylist now is [2 7 10 "Bob" 42]
```

M

map

map [*reporter*] *list* (map [*reporter*] *list1* ... *list2*)

With a single *list*, the given reporter is run for each item in the list, and a list of the results is collected and reported.

In *reporter*, use [?](#) to refer to the current item of *list*.

```
show map [round ?] [1.1 2.2 2.7]
=> [1 2 3]
show map [? * ?] [1 2 3]
=> [1 4 9]
```

With multiple lists, the given reporter is run for each group of items from each list. So, it is run once for the first items, once for the second items, and so on. All the lists must be the same length.

In *reporter*, use [?1](#) through [?n](#) to refer to the current item of each list.

Some examples make this clearer:

```
show (map [?1 + ?2] [1 2 3] [2 4 6])
=> [3 6 9]
show (map [?1 + ?2 = ?3] [1 2 3] [2 4 6] [3 5 9])
=> [true false true]
```

See also [foreach](#), [?](#).

max

max *list*

Reports the maximum number value in the list. It ignores other types of items.

```
show max values-from turtles [xcor]
;; prints the x coordinate of the turtle which is
;; farthest right on the screen
```

max-one-of

max-one-of *agentset* [*reporter*]

Reports the agent in the agentset that has the highest value for the given reporter.

```
show max-one-of patches [count turtles-here]
;; prints the patch with the most turtles on it
```

mean

mean *list*

Reports the statistical mean of the numeric items in the given list. Ignores non-numeric items. The mean is the average, i.e., the sum of the items divided by the total number of items.

```
show mean values-from turtles [xcor]
;; prints the average of all the turtles' x coordinates
```

median

median *list*

Reports the statistical median of the numeric items of the given list. Ignores non-numeric items. The median is the item that would be in the middle if all the items were arranged in order. (If two items would be in the middle, the median is the average of the two.)

```
show median values-from turtles [xcor]
;; prints the median of all the turtles' x coordinates
```

member?

member? *value list* member? *string1 string2*

For a list, reports true if the given value appears in the given list, otherwise reports false.

For a string, reports true or false depending on whether *string1* appears anywhere inside *string2* as a substring.

```
show member? 2 [1 2 3]
=> true
show member? 4 [1 2 3]
=> false
show member? "rin" "string"
=> true
```

See also [position](#).

min

min *list*

Reports the minimum number value in the list. It ignores other types of items.

```
show min values-from turtles [xcor]
;; prints the lowest x-coordinate of all the turtles
```

min-one-of

min-one-of *agentset* [*reporter*]

Reports the agent in the agentset that reports the lowest value for the given reporter.

```
show min-one-of turtles [xcor + ycor]
;; reports the turtle with the smallest sum of
;; its coordinates
```

mod

number1 mod *number2*

Reports *number1* modulo *number2*: that is, the residue of *number1* (mod *number2*). mod is equivalent to the following NetLogo code:

```
number1 - (floor (number1 / number2)) * number2
```

Note that mod is infix, that is, it comes between its two inputs.

```
show 62 mod 5
=> 2
show -8 mod 3
=> 1
```

See also [remainder](#). mod and remainder behave the same for positive numbers, but differently for negative numbers.

mouse-down?

mouse-down?

Reports true if the mouse button is down, false otherwise.

Note: If the mouse pointer is outside of the NetLogo graphics window, `mouse-down?` will always report false.

mouse-xcor

mouse-ycor

mouse-xcor

mouse-ycor

Reports the x or y coordinate of the mouse in the Graphics Window. The value is in terms of turtle coordinates, so it is a floating-point number. If you want patch coordinates, use `round mouse-xcor` and `round mouse-ycor`.

Note: If the mouse is outside of the NetLogo graphics window, reports the value from the last time it was inside.

```
;; to make the mouse "draw" in red:
if mouse-down?
  [ set pcolor-of patch-at mouse-xcor mouse-ycor red ]
```

myself

myself



When an agent has been asked to run some code, using `myself` in that code reports the agent (turtle or patch) that did the asking.

`myself` is most often used in conjunction with `-of` to read or set variables in the asking agent.

`myself` can be used within blocks of code not just in the `ask` command, but also `hatch`, `values-from`, `value-from`, `with`, `min-one-of`, `max-one-of`, and `histogram`.

```
ask turtles
  [ ask patches in-radius 3
    [ set pcolor color-of myself ] ]
;; each turtle makes a colored "splotch" around itself
```

See the `Myself Example` code example for more examples.

See also [self](#).

N

neighbors

neighbors4

neighbors
neighbors4

Reports an agentset containing the 8 surrounding patches (neighbors) or 4 surrounding patches (neighbors4).

```
show sum values-from neighbors [count turtles-here]
  ;; prints the total number of turtles on the eight
  ;; patches around the calling turtle or patch
ask neighbors4 [ set pcolor red ]
  ;; turns the four neighboring patches red
```

no-display**no-display**

Turns off all updating of the graphics window until the display command is issued. This has two major uses.

One, you can control when the user sees screen updates. You might want to change lots of things on the screen behind the users back, so to speak, then make them visible to the user all at once.

Two, your model will run faster when graphics updating is off, so if youre in a hurry, this command will let you get results faster. (Note that normally you dont need to use no-display for this, since you can also use the on/off switch in graphics window control strip to freeze the display.)

Note that display and no-display operate independently of the switch in the graphics window control strip that freezes the display.

See also [display](#).

no-label**no-label**

This is a special value used to remove labels from turtles and patches.

When you set a turtles label to no-label, or a patches plabel to no-label, then a label will no longer be drawn on top of the turtle or patch.

See also [label](#), [label-color](#), [plabel](#), [plabel-color](#).

nobody**nobody**

This is a special value which some primitives such as turtle, random-one-of, max-one-of, etc. return to indicate that no agent was found. Also, when a turtle dies, it becomes equal to nobody.

```
set other random-one-of other-turtles-here
if other != nobody
  [ set color-of other red ]
```

not

not *boolean*

Reports true if *boolean* is false, otherwise reports false.

```
if not (color = blue) [ fd 10 ]
;; all non-blue turtles move forward 10 steps
```

nsum

nsum4

nsum *patch-variable*
nsum4 *patch-variable*



For each patch, reports the sum of the values of *patch-variable* in the 8 surrounding patches (nsum) or 4 surrounding patches (nsum4).

Note that nsum/nsum4 are equivalent to the combination of the sum, values-from, and neighbors/neighbors4 primitives:

```
sum values-from neighbors [var]
  ;; does the same thing as "nsum var"
sum values-from neighbors4 [var]
  ;; does the same thing as "nsum4 var"
```

Therefore nsum and nsum4 are included as separate primitives mainly for backwards compatibility with older versions of NetLogo, which did not have the neighbors and neighbors4 primitives.

See also [neighbors](#), [neighbors4](#).

O

-of

VARIABLE-of *agent*

Reports the value of the *VARIABLE* of the given agent. Can also be used to set the value of the variable.

```
set color-of random-one-of turtles red
;; a randomly chosen turtle turns red
ask turtles [ set pcolor-of (patch-at -1 0) red ]
;; each turtle turns the patch on its left red
```

one-of

one-of *agentset*

If given a turtle agentset, reports the turtle in the set with the lowest numbered ID.

If given a patch agentset, reports the patch in the set with the highest pycor and, if a tie-breaker is needed, with the lowest pxcor.

If the agentset is empty, reports [nobody](#).

See also [random-one-of](#).

or

boolean1 or *boolean2*

Reports true if either *boolean1* or *boolean2*, or both, is true.

Note that if *condition1* is true, then *condition2* will not be run (since it cant affect the result).

```
if (pxcor > 0) or (pycor > 0) [ set pcolor red ]
;; patches turn red except in lower-left quadrant
```

other-turtles-here other-*BREED*-here

other-turtles-here other-*BREED*-here



Reports an agentset consisting of all turtles on the calling turtles patch (*not* including the caller itself). If a breed is specified, only turtles with the given breed are included.

```
;; suppose I am one of 10 turtles on the same patch
show count other-turtles-here
=> 9
```

Example using breeds:

```
breeds [cats dogs]
show count other-dogs-here
;; prints the number of dogs (that are not me) on my patch
```

See also [turtles-here](#).

P

patch

patch *pxcor pycor*

Given two integers, reports the single patch with the given *pxcor* and *pycor*. (The coordinates are the actual coordinates; they are not computed relative to the calling agent, as with `patch-at`.) *pxcor* and *pycor* must be integers.

```
ask (patch 3 -4) [ set pcolor green ]
;; patch with pxcor of 3 and pycor of -4 turns green
```

See also [patch-at](#).

patch-at

patch-at *dx dy*

`patch-at` reports the single patch at (*dx*, *dy*) from the caller, that is, *dx* patches east and *dy* patches north of the caller. (If the caller is the observer, the given offsets are computed from the origin.)

```
ask patch-at 1 -1 [ set pcolor green ]
;; if caller is the observer, turn the patch
;;   at (1, -1) green
;; if caller is a turtle or patch, turns the
;;   patch just southeast of the caller green
```

See also [patch](#).

patch-here

patch-here



`patch-here` reports the patch under the turtle.

Note that this reporter doesn't work for a patch because a patch can just say self.

patches

patches

Reports the agentset consisting of all patches.

patches-own

patches-own [*var1 var2 ...*]

This keyword, like the `globals`, `breeds`, `<BREED>-own`, and `turtles-own` keywords, can only be used at the beginning of a program, before any function definitions. It defines the variables that all patches can use.

All patches will then have the given variables and be able to use them.

See also [globals](#), [turtles-own](#), [breeds](#), [<BREED>-own](#).

pcolor

pcolor



This is a built-in patch variable. It holds the color of the patch. You can set this variable to make the patch change color.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [color](#).

pen-down

pd

pen-up

pu

pen-down

pen-up



The turtle its pen down (or up), so that it draws (leaves a trail) when they move (or doesnt).

Turtles draw by changing the color of the patches underneath them to their own color. To change the color of the turtles pen (and the color of the turtle itself), use `set color`.

Note: When a turtles pen is down, only the commands `forward` and `back` cause drawing.

Note: Theses commands are equivalent to setting the turtle variable `pen-down?` to true or false.

See also [pen-down?](#).

pen-down?

pen-down?



This is a built-in turtle variable. It holds a boolean (true or false) value indicating whether the turtles pen is currently down. You can set this variable to put a turtles pen down or pick it back up again.

See also [pen-down](#), [pen-up](#).

plabel

plabel



This is a built-in patch variable. It may hold a value of any type. The patch appears in the graphics window with the given value attached to it as text. You can set this variable to add, change, or remove a patches label.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [no-label](#), [plabel-color](#), [label](#), [label-color](#).

plabel-color

plabel-color



This is a built-in patch variable. It holds a number greater than or equal to 0 and less than 140. This number determines what color the patches label appears in (if it has a label). You can set this variable to change the color of a patches label.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [no-label](#), [plabel](#), [label](#), [label-color](#).

plot

plot *number*

Increments the x-value of the plot pen by `plot-pen-interval`, then plots a point at the updated x-value and a y-value of *number*. (The first time the command is used on a plot, the point plotted has an x-value of 0.)

plot-name

plot-name

Reports the name of the current plot (a string).

plot-pen-down

ppd

plot-pen-up

ppu

plot-pen-down
plot-pen-up

Puts down (or up) the current plot-pen, so that it draws (or doesn't). (By default, all pens are down initially.)

plot-pen-reset**plot-pen-reset**

Clears everything the current plot pen has drawn, moves it to (0,0), and puts it down. If the pen is a permanent pen, the color and mode are reset to the default values from the plot Edit dialog.

plotxy**plotxy *number1 number2***

Moves the current plot pen to the point with coordinates (*number1*, *number2*). If the pen is down, a line, bar, or point will be drawn (depending on the pen's mode).

plot-x-min
plot-x-max
plot-y-min
plot-y-max**plot-x-min**
plot-x-max
plot-y-min
plot-y-max

Reports the minimum or maximum value on the x or y axis of the current plot.

These values can be set with the commands `set-plot-x-range` and `set-plot-y-range`. (Their default values are set from the plot Edit dialog.)

position**position *item list***
position *string1 string2*

On a list, reports the first position of *item* in *list*, or false if it does not appear.

On strings, reports the position of the first appearance *string1* as a substring of *string2*, or false if it does not appear.

Note: The positions are numbered beginning with 0, not with 1.

```
;; suppose mylist is [2 7 4 7 "Bob"]
show position 7 mylist
```

```
=> 1
show position 10 mylist
=> false
show position "rin" "string"
=> 2
```

See also [member?](#).

precision

precision *number places*

Reports *number* rounded to *places* decimal places.

If *places* is negative, the rounding takes place to the left of the decimal point.

```
show precision 1.23456789 3
=> 1.235
show precision 3834 -3
=> 4000
```

print

print *value*

Prints *value* in the Command Center, followed by a carriage return.

The calling agent is *not* printed before the value, unlike [show](#).

See also [show](#), [type](#).

pxcor

pxcor



This is a built-in patch variable. It holds the x coordinate of the patch. This is always an integer. You cannot set this variable, because patches don't move.

This variable is greater than or equal to 0 – screen-edge-x and less than or equal to screen-edge-x.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [pycor](#), [xcor](#), [ycor](#).

pycor

pycor

This is a built-in patch variable. It holds the y coordinate of the patch. This is always an integer. You cannot set this variable, because patches don't move.

This variable is greater than or equal to 0 – screen-edge-y and less than or equal to screen-edge-y.

All patch variables can be directly accessed by any turtle standing on the patch.

See also [pxcor](#), [xcor](#), [ycor](#).

R

random

random number

If *number* is positive, reports a random number greater than or equal to 0 but strictly less than *number*.

If *number* is negative, the number reported is less than or equal to 0, but strictly greater than *number*.

If *number* is zero, the result is always zero as well.

If *number* is an integer, reports a random integer.

If *number* is floating point (has a decimal point), reports a floating point number.

```
show random 3
;; prints 0, 1, or 2
show random 5.0
;; prints a number at least 0.0 but less than 5.0,
;; for example 4.686596634174661
```

random-exponential

random-normal

random-poisson

random-exponential mean

random-normal mean standard-deviation

random-poisson mean

Reports an accordingly distributed random number with the *mean* and, in the case of the normal distribution, the *standard-deviation*.

random-exponential reports an exponentially distributed random floating point number.

`random-normal` reports a normally distributed random floating point number.

`random-poisson` reports a Poisson-distributed random integer.

```
show random-exponential 2
;; prints an exponentially distributed random floating
;; point number with a mean of 2
show random-normal 10.1 5.2
;; prints a normally distributed random floating point
;; number with a mean of 10.1 and a standard deviation
;; of 5.2
show random-poisson 3.4
;; prints a Poisson-distributed random integer with a
;; mean of 3.4
```

random-n-of

random-n-of *size agentset*

Reports an agentset of size *size* randomly chosen from the input agentset. It is an error for *size* to be greater than the size of the supplied agentset.

```
ask random-n-of 50 patches [ set pcolor green ]
;; 50 random patches turn green
```

See also [random-one-of](#).

random-one-of

random-one-of *agentset*

Reports a random agent chosen from the agentset. If the agentset is empty, reports [nobody](#).

```
ask random-one-of patches [ set pcolor green ]
;; a random patch turns green
set pcolor-of random-one-of patches green
;; another way to say the same thing
ask patches with [any turtles-here]
  [ show random-one-of turtles-here ]
;; for each patch containing turtles, prints one of
;; those turtles
```

See also [one-of](#) and [random-n-of](#).

random-seed

random-seed *number*

Sets the seed of the pseudo-random number generator to the integer part of *number*. The exact value of the seed does not matter except as indicated below. The seed may be any integer in the range supported by NetLogo (−2147483648 to 2147483647).

If the pseudo-random number generator is initialized to a fixed seed, then it will generate exactly

the same random sequence every time a model is run.

On the other hand, setting the seed to different values in each run will make the sequences different for each run.

When this command is not used, the pseudo-random number generator uses a seed based on the current time.

```
random-seed 47823
show random 100
=> 57
show random 100
=> 91
random-seed 47823
show random 100
=> 57
show random 100
=> 91
```

read-from-string

read-from-string *string*

Interprets the given string as if it had been typed in the Command Center, and reports the resulting value. The result may be a number, list, string, or boolean value, or the special value nobody.

Useful in conjunction with the [user-input](#) primitive for converting the users input into usable form.

```
show read-from-string "3" + read-from-string "5"
=> 8
show length read-from-string "[1 2 3]"
=> 3
crt read-from-string user-input "Make how many turtles?"
;; the number of turtles input by the user
;; are created
```

reduce

reduce [*reporter*] *list*

Reduces a list from left to right using *reporter*, resulting in a single value. This means, for example, that `reduce [?1 + ?2] [1 2 3 4]` is equivalent to $((1 + 2) + 3) + 4$. If *list* has a single item, that item is returned. It is an error to reduce an empty list.

In *reporter*, use ?1 and ?2 to refer to the two objects being combined.

Since it can be difficult to develop an intuition about what `reduce` does, here are some simple examples which, while not useful in themselves, may give you a better understanding of this primitive:

```
show reduce [?1 + ?2] [1 2 3]
=> 6
show reduce [?1 - ?2] [1 2 3]
=> -4
```

```

show reduce [?2 - ?1] [1 2 3]
=> 2
show reduce [?1] [1 2 3]
=> 1
show reduce [?2] [1 2 3]
=> 3
show reduce [sentence ?1 ?2] [[1 2] [3 [4]] 5]
=> [1 2 3 [4] 5]
show reduce [fput ?2 ?1] (fput [] [1 2 3 4 5])
=> [5 4 3 2 1]

```

Here are some more useful examples:

```

to-report longer-string [str1 str2]
  ifelse length str1 >= length str2
    [ report str1 ]
    [ report str2 ]
end

;; find the longest string in a list
to-report longest-string [strings]
  report reduce [longer-string ?1 ?2] strings
end

show longest-string ["hi" "there" "!"]
=> "there"

;; evaluate the polynomial, with given coefficients, at x
to-report eval-polynomial [coeffs x]
  report reduce [(x * ?1) + ?2] coeffs
end

;; evaluate 3x^2 + 2x + 1 at x = 4
show eval-polynomial [3 2 1] 4
=> 57

```

remainder

remainder *number1 number2*

Reports the remainder when *number1* is divided by *number2*. This is equivalent to the following NetLogo code:

```

number1 - (int (number1 / number2)) * number2

show remainder 62 5
=> 2
show remainder -8 3
=> -2

```

See also [mod](#). `mod` and `remainder` behave the same for positive numbers, but differently for negative numbers.

remove

remove *item list*

For a list, reports a copy of *list* with all instances of *item* removed.

For strings, reports a copy of *string2* with all the appearances of *string1* as a substring removed.

```

set mylist [2 7 4 7 "Bob"]
set mylist remove 7 mylist
;; mylist is now [2 4 "Bob"]
show remove "na" "banana"
=> "ba"

```

remove-duplicates

remove-duplicates *list*

Reports a copy of *list* with all duplicate items removed. The first of each item remains in place.

```

set mylist [2 7 4 7 "Bob" 7]
set mylist remove-duplicates mylist
;; mylist is now [2 7 4 "Bob"]

```

repeat

repeat *number* [*commands*]

Runs *commands* *number* times.

```

pd repeat 36 [ fd 1 rt 10 ]
;; the turtle draws a circle

```

replace-item

replace-item *index list value* replace-item *index string1 string2*

On a list, replaces an item in that list. *index* is the index of the item to be replaced, starting with 0. (The 6th item in a list would have an index of 5.) Note that `replace-item` is used in conjunction with `set` to change a list.

Likewise for a string, but the given character of *string1* removed and the contents of *string2* spliced in instead.

```

show replace-item 2 [2 7 4 5] 15
=> [2 7 15 5]
show replace-item 1 "sat" "lo"
=> "slot"

```

report

report *value*

Immediately exits from the current `to-report` procedure and reports *value* as the result of that procedure. `report` and `to-report` are always used in conjunction with each other. See [to-report](#) for a discussion of how to use them.

reset-timer

reset-timer

Resets the global clock to zero. See also [timer](#).

reverse

reverse *list*

reverse *string*

Reports a reversed copy of the given list or string.

```
show mylist
;; mylist is [2 7 4 "Bob"]
set mylist reverse mylist
;; mylist now is ["Bob" 4 7 2]
show reverse "string"
=> "gnirts"
```

rgb

rgb *red green blue*

Reports a number in the range 0 to 140, not including 140 itself, that represents the given color, specified in the RGB spectrum, in NetLogo's color space.

All three inputs should be in the range 0.0 to 1.0.

The color reported may be only an approximation, since the NetLogo color space does not include all possible colors. (See [hsb](#) for a description of what parts of the HSB color space NetLogo colors cover; this is difficult to characterize in RGB terms.)

```
show rgb 0 0 0
=> 0.0 ;; black
show rgb 0 1.0 1.0
=> 85.0 ;; cyan
```

See also [extract-rgb](#), [hsb](#), and [extract-hsb](#).

right rt

right *number*



The turtle turns right by *number* degrees. (If *number* is negative, it turns left.)

round

round *number*

Reports the integer nearest to *number*.

If the decimal portion of *number* is exactly .5, the number is rounded in the **positive** direction.

Note that rounding in the positive direction is not always how rounding is done in other software programs. (In particular, it does not match the behavior of the StarLogoT, which always rounded numbers ending in 0.5 to the nearest even integer.) The rationale for this behavior is that it matches how turtle coordinates relate to patch coordinates in NetLogo. For example, if a turtles xcor is -4.5 , then it is on the boundary between a patch whose pxcor is -4 and a patch whose pxcor is -5 , but the turtle must be considered to be in one patch or the other, so the turtle is considered to be in the patch whose pxcor is -4 , because we round towards the positive numbers.

```
show round 4.2
=> 4
show round 4.5
=> 5
show round -4.5
=> -4
```

run

run *string*

This agent interprets the given string as a sequence of one or more NetLogo commands and runs them.

The code runs in the agents current context, which means it has access to the values of local variables, myself, and so on.

See also [runresult](#).

runresult

runresult *string*

This agent interprets the given string as a NetLogo reporter and runs it, reporting the result obtained.

The code runs in the agents current context, which means it has access to the values of local variables, myself, and so on.

See also [run](#).

S

scale-color

scale-color *color number range1 range2*

Reports a shade of *color* proportional to *number*.

If *range1* is less than *range2*, then the larger the number, the lighter the shade of *color*. But if *range2* is less than *range1*, the color scaling is inverted.

If *number* is less than *range1*, then the darkest shade of *color* is chosen.

If *number* is greater than *range2*, then the lightest shade of *color* is chosen.

Note: for *color* shade is irrelevant, e.g. green and green + 2 are equivalent, and the same spectrum of colors will be used.

```
ask turtles [ set color scale-color red age 0 50 ]
;; colors each turtle a shade of red proportional
;; to its value for the age variable
```

screen-edge-x

screen-edge-y

screen-edge-x

screen-edge-y

These reporters give the maximum x-coordinate and maximum y-coordinate (respectively) of the Graphics Window.

screen-edge-x and -y are the half-width and half-height of the NetLogo world the distances from the origin to the edges. screen-size is the same as ((2 * screen-edge) + 1).

Note: You can set the size of the Graphics Window only by editing it these are reporters which cannot be set.

```
cct 100 [ setxy (random screen-edge-x)
               (random screen-edge-y) ]
;; distributes 100 turtles randomly in the
;; first quadrant
```

screen-size-x screen-size-y

screen-size-x screen-size-y

These reporters give the total width and height of the NetLogo world.

Screen-size is the same as $((2 * \text{screen-edge}) + 1)$.

self

self



Reports this turtle or patch.

```
ask turtles with [self != myself]
  [ die ]
;; this turtle kills all other turtles
```

See also [myself](#).

; (semicolon)

; *comments*

After a semicolon, the rest of the line is ignored. This is useful for adding comments to your code text that explains the code to human readers. Extra semicolons can be added for visual effect.

sentence

se

sentence *value1 value2* (sentence *value1 ... valuen*)

Makes a list out of the values. If any value is a list, its items are included in the result directly, rather than being included as a sublist. Examples make this clearer:

```
show sentence 1 2
=> [1 2]
show sentence [1 2] 3
=> [1 2 3]
show sentence 1 [2 3]
=> [1 2 3]
show sentence [1 2] [3 4]
=> [1 2 3 4]
show (sentence [1 2] 3 [4 5] (3 + 3) 7)
=> [1 2 3 4 5 6 7]
```

set

set *variable value*

Sets *variable* to the given value.

Variable can be any of the following:

- An global variable declared using `globals`
- The global variable associated with a slider, switch, or choice
- A variable belonging to the calling agent
- If the calling agent is a turtle, a variable belonging to the patch under the turtle.
- An expression of the form *VARIABLE-of agent*

set-current-plot

set-current-plot *plotname*

Sets the current plot to the plot with the given name (a string). Subsequent plotting commands will affect the current plot.

set-current-plot-pen

set-current-plot-pen *penname*

The current plots current pen is set to the pen named *penname* (a string). If no such pen exists in the current plot, a runtime error occurs.

set-default-shape

set-default-shape turtles *string*

set-default-shape *breed string*



Specifies a default initial shape for all turtles, or for a particular breed. When a turtle is created, or it changes breeds, its shape is set to the given shape.

The specified breed must be either `turtles` or a breed defined by the [breeds](#) keyword, and the specified string must be the name of a currently defined shape.

In new models, the default shape for all turtles is `default`.

Note that specifying a default shape does not prevent you from changing an individual turtle's shape later; turtles don't have to be stuck with their breed's default shape.

```
create-turtles 1 ;; new turtle's shape is "default"
create-cats 1   ;; new turtle's shape is "default"

set-default-shape turtles "circle"
create-turtles 1 ;; new turtle's shape is "circle"
```

```

create-cats 1    ;; new turtle's shape is "circle"

set-default-shape cats "cat"
set-default-shape dogs "dog"
create-cats 1    ;; new turtle's shape is "cat"
ask cats [ set breed dogs ]
    ;; all cats become dogs, and automatically
    ;; change their shape to "dog"

```

See also [shape](#).

set-histogram-num-bars

set-histogram-num-bars *integer*

Set the current plot pens plot interval so that, given the current x range for the plot, there would be *integer* number of bars drawn if the histogram or histogram-list commands were called.

See also [histogram](#).

set-plot-pen-color

set-plot-pen-color *number*

Sets the color of the current plot pen to *number*.

set-plot-pen-interval

set-plot-pen-interval *number*

Tells the current plot pen to move a distance of *number* in the x direction during each use of the plot command. (The plot pen interval also affects the behavior of the histogram and histogram-list commands.)

set-plot-pen-mode

set-plot-pen-mode *number*

Sets the mode the current plot pen draws in to *number*. The allowed plot pen modes are:

- 0 (line mode) the plot pen draws a line connecting two points together.
- 1 (bar mode): the plot pen draws a bar of width plot-pen-interval with the point plotted as the upper (or lower, if you are plotting a negative number) left corner of the bar.
- 2 (point mode): the plot pen draws a point at the point plotted. Points are not connected.

The default mode for new pens is 0 (line mode).

set-plot-x-range

set-plot-y-range

set-plot-x-range *min max*
set-plot-y-range *min max*

Sets the minimum and maximum values of the x or y axis of the current plot.

The change is temporary and is not saved with the model. When the plot is cleared, the ranges will revert to their default values as set in the plots Edit dialog.

setxy

setxy *x y*



The turtle sets its x-coordinate to *x* and its y-coordinate to *y*.

Equivalent to `set xcor x set ycor y`, except it happens in one time step instead of two.

```
setxy 0 0
;; turtle moves to the middle of the center patch
```

shade-of?

shade-of? *color1 color2*

Reports true if both colors are shades of one another, false otherwise.

```
show shade-of? blue red
=> false
show shade-of? blue (blue + 1)
=> true
show shade-of? gray white
=> true
```

shape

shape



This is a built-in turtle variable. It holds a string that is the name of the turtles current shape. You can set this variable to change a turtles shape. New turtles have the shape default unless the a different shape has been specified using [set-default-shape](#).

See also [set-default-shape](#).

show

show *value*

Prints *value* in the Command Center, preceded by the calling agent, and followed by a carriage return. (The calling agent is included to help you keep track of what agents are producing which lines of output.)

See also [print](#), [type](#).

showturtle

st

showturtle



The turtle becomes visible again.

Note: This command is equivalent to setting the turtle variable `hidden?` to `false`.

See also [hideturtle](#).

sin

sin *number*

Reports the sine of the given angle. Assumes angle is given in degrees.

```
show sin 270
=> -1.0
```

size

size



This is a built-in turtle variable. It holds a number that is the turtles apparent size in the graphics window. The default size for a turtle is 1.0, which means that the turtle is the same size as a patch. You can set this variable to change a turtles size.

All turtles appear the same size in the graphics window unless the Turtle Sizes checkbox in the graphics window edit dialog is checked. If that checkbox is not checked, you can still use this variable, but it will not have any visible effect.

Note: the Turtle Sizes feature is currently considered experimental. It may cause your model to run much more slowly, and it may cause display anomalies.

sort

sort *list*

Reports a new list containing the same items as the input list, but in ascending order.

If there is at least one number in the list, the list is sorted in numerically ascending order and any non-numeric items of the input list are discarded.

If there are no numbers, but at least one string in the list, the list is sorted in alphabetically ascending order and any non-string items are discarded.

sort-by

sort-by [*reporter*] *list*

Reports a new list containing the same items as the input list, in a sorted order defined by the boolean (true or false) *reporter*.

In *reporter*, use ?1 and ?2 to refer to the two objects being compared. *reporter* should be true if ?1 comes strictly before ?2 in the desired sort order, and false otherwise.

```
show sort-by [?1 < ?2] [3 1 4 2]
=> [1 2 3 4]
show sort-by [?1 > ?2] [3 1 4 2]
=> [4 3 2 1]
show sort-by [length ?1 < length ?2] ["zzz" "z" "zz"]
=> [z zz zzz]
show sort-by [random 2 = 0] [1 2 3 4]
=> [2 3 1 4] ; ; or other random order
```

sprout

sprout *number* [*commands*]



Creates *number* new turtles on the current patch. The new turtles have random colors and orientations, and they immediately run *commands*. This is useful for giving the new turtles different colors, headings, breeds, or whatever.

```
sprout 1 [ set color red ]
```

Note: While the commands are running, no other agents are allowed to run any code (as with the without-interruption command). This ensures that the new turtles cannot interact with any other agents until they are fully initialized. In addition, no screen updates take place until the commands are done. This ensures that the new turtles are never drawn on-screen until they are fully initialized.

sqrt

sqrt *number*

Reports the square root of *number*.

stamp

stamp *color*



Sets the color of the patch under the turtle to the given color.

```
repeat 30 [ stamp yellow fd 3 rt 6 ]
;; the turtle records its arched path contrast to the
;; effect of "pen-down"
```

standard-deviation

standard-deviation *list*

Reports the unbiased statistical standard deviation of a *list* of numbers. Ignores other types of items.

```
show standard-deviation [1 2 3 4 5 6]
=> 1.8708286933869707
show standard-deviation values-from turtles [energy]
;; prints the standard deviation of the variable "energy"
;; from all the turtles
```

startup

startup



User-defined procedure which, if it exists, will be called when a model is first loaded.

```
to startup
  setup
end
```

stop

stop

The calling agent exits immediately from the enclosing procedure, ask, or ask-like construct (cct, hatch, sprout). Only the current procedure stops, not all execution for the agent.

Note: stop can be used to stop a forever button. If the forever button directly calls a procedure, then when that procedure stops, the button stops. (In a turtle or patch forever button, the button wont stop until every turtle or patch stops a single turtle or patch doesnt have the power to stop the whole button.)

substring

substring *string position1 position2*

Reports just a section of the given string, ranging between the given positions.

Note: The positions are numbered beginning with 0, not with 1.

```
show substring "turtle" 1 4
=> "urt"
```

sum

sum *list*

Reports the sum of the items in the list.

```
show sum values-from turtles [energy]
;; prints the total of the variable "energy"
;; from all the turtles
```

T

tan

tan *number*

Reports the tangent of the given angle. Assumes the angle is given in degrees.

timer

timer

Reports how many seconds have passed since the command [reset-timer](#) was last run (or since NetLogo started). The potential resolution of the clock is milliseconds. (Whether you get resolution that high in practice may vary from system to system, depending on the capabilities of the underlying Java Virtual Machine.)

to

to *procedure-name*

to *procedure-name* [*input1 input2 ...*]

Used to begin a command procedure.

```
to setup
  ca
  crt 500
end
```

```
to circle [radius]
  cct 100 [ fd radius ]
end
```

to-report

to-report *procedure-name*

to-report *procedure-name* [*input1 input2 ...*]

Used to begin a reporter procedure.

The body of the procedure should use `report` to report a value for the procedure. See [report](#).

```
to-report average [a b]
  report (a + b) / 2
end
```

```
to-report absolute-value [number]
  ifelse number >= 0
    [ report number ]
    [ report 0 - number ]
end
```

```
to-report first-turtle?
  report who = 0 ;; reports true or false
end
```

towards

towards-nowrap

towards *agent*

towards-nowrap *agent*



Reports the heading from this agent to the given agent.

If the wrapped distance (around the edges of the screen) is shorter than the on-screen distance, `towards` will report the heading of the wrapped path. `towards-nowrap` never uses the wrapped path.

towardsxy

towardsxy-nowrap

towardsxy *x y*

towardsxy-nowrap *x y*



Reports the heading from the turtle or patch towards the point (x,y).

If the wrapped distance (around the edges of the screen) is shorter than the on-screen distance, `towardsxy` will report the heading of the wrapped path. `towardsxy-nowrap` never uses the wrapped path.

turtle

turtle *number*

Reports the turtle with the given ID number, or [nobody](#) if there is no such turtle. *number* must be an integer.

```
set color-of turtle 5 red
;; turtle with id number 5 turns red
ask turtle 5 [ set color red ]
;; another way to do the same thing
```

turtles

turtles

Reports the agentset consisting of all turtles.

```
show count turtles
;; prints the number of turtles
```

turtles-at **BREED-at**

turtles-at *dx dy* **BREED-at *dx dy***

Reports an agentset containing the turtles on the patch (dx, dy) from the caller (including the caller itself if its a turtle). If the caller is the observer, dx and dy are calculated from the origin (0,0).

```
;; suppose I have 40 turtles at the origin
show count turtles-at 0 0
=> 40
```

If the name of a breed is substituted for turtles, then only turtles of that breed are counted.

```
breeds [cats dogs]
create-custom-dogs 5 [ setxy 2 3 ]
show count dogs-at 2 3
=> 5
```

turtles-here **BREED-here**

turtles-here **BREED-here**

Reports an agentset containing all the turtles on the callers patch (including the caller itself if its a turtle).

```
ca
```

```
crt 10
ask turtle 0 [ show count turtles-here ]
=> 10
```

If the name of a breed is substituted for turtles, then only turtles of that breed are counted.

```
breeds [cats dogs]
create-cats 5
create-dogs 1
ask dogs [ show count cats-here ]
=> 5
```

See also [other-turtles-here](#).

turtles-own *BREED*-own

turtles-own [*var1 var2 ...*]
***BREED*-own** [*var1 var2 ...*]

The turtles-own keyword, like the globals, breed, <*BREED*>-own, and patches-own keywords, can only be used at the beginning of a program, before any function definitions. It defines the variables belonging to each turtle.

If you specify a breed instead of turtles, only turtles of that breed have the listed variables. (More than one breed may list the same variable.)

```
breeds [cats dogs hamsters]
turtles-own [eyes legs] ;; applies to all breeds
cats-own [fur kittens]
hamsters-own [fur cage]
dogs-own [hair puppies]
```

See also [globals](#), [patches-own](#), [breeds](#), [<BREED>-own](#).

type

type *value*

Prints *value* in the Command Center, *not* followed by a carriage return (unlike [print](#) and [show](#)). The lack of a carriage return allows you to print several values on the same line.

The calling agent is *not* printed before the value. unlike [show](#).

```
type 3 type " " print 4
=> 3 4
```

See also [print](#), [show](#).

U

uphill

uphill *patch-variable*



Reports the turtle heading (between 0 and 359 degrees) in the direction of the maximum value of the variable *patch-variable*, of the patches in a one-patch radius of the turtle. (This could be as many as eight or as few as five patches, depending on the position of the turtle within its patch.)

If there are multiple patches that have the same greatest value, a random one of those patches will be selected.

If the patch is located directly to the north, south, east, or west of the patch that the turtle is currently on, a multiple of 90 degrees is reported. However, if the patch is located to the northeast, northwest, southeast, or southwest of the patch that the turtle is currently on, the direction the turtle would need to reach the nearest corner of that patch is reported.

See also [uphill4](#), [downhill](#), [downhill](#).

uphill4

uphill4 *patch-variable*



Reports the turtle heading (between 0 and 359 degrees) as a multiple of 90 degrees in the direction of the maximum value of the variable *patch-variable*, of the four patches to the north, south, east, and west of the turtle. If there are multiple patches that have the same greatest value, a random patch from those patches will be selected.

See also [uphill](#), [downhill](#), [downhill4](#).

user-choice

user-choice *value list-of-choices*

Opens a dialog with *value* displayed as the message and a button corresponding to each item in *list-of-choices*.

Reports the item in *list-of-choices* that is associated with the button the user presses.

value may be of any type, but is typically a string.

```
if "yes" = (user-choice "Do you really want to setup the model?" [ "no" "yes" ])
  [ setup ]
```

user-input

user-input *value*

Reports the string that a user types into an entry field in a dialog with title *value*.

value may be of any type, but is typically a string.

```
show user-input "What is your name?"
```

user-message**user-message *value***

Opens a dialog with *value* displayed as the message.

value may be of any type, but is typically a string.

```
user-message "There are " + count turtles + " turtles."
```

user-yes-or-no**user-yes-or-no *value***

Reports true or false based on the users response to *value*.

value may be of any type, but is typically a string.

```
if user-yes-or-no "Do you really want to set up the model?"
  [ setup ]
```

V**value-from****value-from *agent* [*reporter*]**

Reports the value of the reporter for the given agent (turtle or patch).

```
show value-from (turtle 5) [who * who]
=> 25
show value-from (patch 0 0) [count turtles in-radius 3]
;; prints the number of turtles located within a
;; three-patch radius of the origin
```

values-from**values-from *agentset* [*reporter*]**

Reports a list that contains the value of the reporter for each agent in the agentset.

```
ca
crt 4
```

```
show values-from turtles [who]
=> [0 1 2 3]
show values-from turtles [who * who]
=> [0 1 4 9]
```

variance

variance *list*

Reports the sample variance of a *list* of numbers. Ignores other types of items.

The sample variance is the sum of the squares of the deviations of the numbers from their mean, divided by one less than the number of numbers in the list.

```
show variance [2 7 4 3 5]
=> 3.7
```

W

wait

wait *number*

Wait the given number of seconds. (You can use floating-point numbers to specify fractions of seconds.) Note that you can't expect complete precision; the agent will never wait less than the given amount, but might wait slightly more.

```
repeat 10 [ fd 1 wait 0.5 ]
```

See also [every](#).

while

while [*reporter*] [*commands*]

If *reporter* reports false, exit the loop. Otherwise run *commands* and repeat.

The reporter may have different values for different agents, so some agents may run *commands* a different number of times than other agents.

```
while [any other-turtles-here]
  [ fd 1 ]
;; turtle moves until it finds a patch that has
;; no other turtles on it
```

who

who



This is a built-in turtle variable. It holds the turtles id number (an integer greater than or equal to zero). You cannot set this variable; a turtles id number never changes.

When NetLogo starts, or after you use the [clear-all](#) or [clear-turtles](#) commands, new turtles are created with ids in order, starting at 0. If a turtle dies, though, a new turtle may eventually be assigned the same id number that was used by the dead turtle.

Example:

```
show values-from (turtles with [color = red]) [who]
;; prints a list of the id numbers of all red turtles
;; in the Command Center
ca
cct 100
  [ ifelse who <50
    [ set color red ]
    [ set color blue ] ]
;; turtles 0 through 49 are red, turtles 50
;; through 99 are blue
```

You can use the turtle reporter to retrieve a turtle with a given id number. See also [turtle](#).

with

agentset with [reporter]

Takes two inputs: on the left, an agentset (usually turtles or patches). On the right, a boolean reporter. Reports a new agentset containing only those agents that reported true in other words, the agents satisfying the given condition.

```
show count patches with [pcolor = red]
;; prints the number of red patches
```

without-interruption

without-interruption [commands]

The agent runs all the commands in the block without allowing other agents to interrupt. That is, other agents are put on hold and do not execute any commands until the commands in the block are finished.

```
crt 5
ask turtles
  [ without-interruption
    [ type 1 fd 1 type 2 ] ]
=> 1212121212
;; because each turtle will output 1 and move, then output 2
;; however:
ask turtles
  [ type 1 fd 1 type 2 ]
=> 1111122222
;; because each turtle will output 1 and move, then output 2
```

word

word *value1 value2*
(word *value1 ... valuen***)**

Concatenates the inputs together and reports the result as a string.

```
show word "tur" "tle"
=> "turtle"
word "a" 6
=> "a6"
set directory "c:\\foo\\fish\\"
show word directory "bar.txt"
=> "c:\foo\fish\bar.txt"
show word [1 54 8] "fishy"
=> "[1 54 8]fishy"
show (word "a" "b" "c" 1 23)
=> "abc123"
```

wrap-color

wrap-color *number*

`wrap-color` checks whether *number* is in the NetLogo color range of 0 to 140 (not including 140 itself). If it is not, `wrap-color` wraps the numeric input to the 0 to 140 range.

The wrapping is done by repeatedly adding or subtracting 140 from the given number until it is in the 0 to 140 range. (This is the same wrapping that is done automatically if you assign an out-of-range number to the color turtle variable or `pcolor` patch variable.)

```
show wrap-color 150
=> 10
show wrap-color -10
=> 130
```

X

xcor

xcor



This is a built-in turtle variable. It holds the current x coordinate of the turtle. This is a floating point number, not an integer. You can set this variable to change the turtles location.

This variable is always greater than or equal to 0 – `screen-edge-x` and strictly less than `screen-edge-x`.

See also [setxy](#), [ycor](#), [pxcor](#), [pycor](#),

XOR

boolean1 xor boolean2

Reports true if either *boolean1* or *boolean2* is true, but not when both are true.

```
if (pxcor > 0) xor (pycor > 0)
  [ set pcolor blue ]
;; upper-left and lower-right quadrants turn blue
```

Y

ycor

ycor



This is a built-in turtle variable. It holds the current y coordinate of the turtle. This is a floating point number, not an integer. You can set this variable to change the turtles location.

This variable is always greater than or equal to 0 – screen-edge-y and strictly less than screen-edge-y.

See also [setxy](#), [xcor](#), [pxcor](#), [pycor](#),

?

?

?, ?1, ?2, ...

These are special local variables. They hold the current inputs to a reporter or command block for certain primitives (for example, the current item of a list being visited by [foreach](#) or [map](#)).

? is always equivalent to ?1.

You may not set these variables, and you may not use them except with certain primitives, currently [foreach](#), [map](#), [reduce](#), [filter](#), and [sort-by](#). See those entries for example usage.

